AFIT/GCS/ENG/93D-22

Object Interaction in a Parallel Object-Oriented Discrete-Event Simulation

THESIS

Walter Gordon Trachsel
Captain, United States Air Force

AFIT/GCS/ENG/93D-22

Approved for public release; distribution unlimited

AFIT/GCS/ENG/93D-22

Object Interaction in a

Parallel Object-Oriented Discrete-Event Simulation

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Walter Gordon Trachsel, M.B.A, B.S.C.S.

Captain, United States Air Force

15 December 1993

Acknowledgements

I would like to thank my advisor, Dr. Hartrum, for his guidance and patience during this research effort. I would also like to thank my original committee members, MAJ Christensen and MAJ Sonnier for their help at the beginning of the research, and then Lt Col Amburn and Major Luginbuhl for joining my committee late in the research when the other two members left AFIT.

Most importantly, I would like to thank my wife Joy for her patience and love during this entire thesis experience when I had to be working almost all the time. I would also thank my daughter Ashley for understanding why I could not spend more time with her. Finally, I would like to thank God for giving me my new daughter Allison, who was born just days before completion of this thesis, to lift me up and keep me going.

Walter Gordon Trachsel

Table of Contents

List of Figures

List of Tables

Abstract

This thesis investigates object interaction issues involved in developing an object-oriented parallel discrete-event simulation and develops a simulation model that provides object interaction capabilities. The research covers issues in object representation, object interaction, object management, discrete-event simulation, and parallel simulation.

There are three primary types of objects that the research discusses. The first type is a basic simulation object, whose size and behavior is insignificant compared to the size of the simulation as a whole. The second type is an aggregate object which consists of smaller component objects that interact and affect the performance of the larger object as a whole. Finally, there are environmental objects, such as terrain and weather, whose size and impact are significant to the entire simulation environment. This research addresses issues about whether these objects can be represented the same in the simulation or whether a special case must exist for each type of object.

As a result of the research, a simulation model is designed that allows interaction between simulation objects. The design goals used to develop the model were strict use of object-oriented practices, ease of simulation modification, and reuse across other applications in the same class of simulations. In this case, the class of simulations is interacting objects in a spatially partitioned discrete-event simulation.

The model is then implemented using AFIT's Battlesim program, which is a parallel discrete-event simulation that has a battlefield divided into partitions and allows objects to move and interact.

Object Interaction in a Parallel Object-Oriented Discrete-Event Simulation

I. Background and Statement of the Problem

1.1 Introduction

This thesis investigates the application of object-oriented programming to parallel computer simulation. It specifically addresses the interaction between simulation objects in a parallel discrete-event simulation environment.

The purpose of this chapter is to provide a background in the research area, define the problem, and explain how the problem was approached. This chapter begins with a background of the research topic and explains the problem that the research addresses. Then it gives a summary of current knowledge of the problem and defines the scope of the study. Following this explanation of the problem, the chapter addresses the standards that were used and the approach that was followed.

1.2 Background

1.2.1 Object-Oriented Simulation. The current trend in simulation is to use object-oriented techniques for modeling real-world objects and processes. These techniques are new to parallel simulation, particularly in the areas of object management and object interaction. Object management is the process of keeping track of the simulation objects and controlling their creation, access, and destruction. Object interaction is the process of the objects communicating with each other and reacting to the communication. This interaction can either be direct or through the management of the simulation support system.

The technique of modeling a problem as a set objects and their relationship to each other makes the object-oriented paradigm a good candidate for use in simulation because each object in a simulation is a direct representation of a real-world object. One advantage of using the object-oriented paradigm is that it usually results in the development of well-structured software. The good program structure is a direct result of the use of object data encapsulation, which requires all the information and program data for each object to be contained in one place. Well-structured programs are much easier to maintain than programs that are poorly structured. Another advantage of object-oriented programming in simulation is that it promotes reusability of software. Objects that are defined for specific simulations may be reused in other simulations that require the same type of objects (3:195).

Like any other programming method, object-oriented programming requires the steps of analysis, design, and implementation. Object-oriented analysis is a well-defined process used to analyze and define a system to be modeled. However, the decisions made during object-oriented design and implementation will most significantly determine the performance of the final application. Simple object interaction is easily defined during the analysis stage. For example, Figure 1 shows the simple object representation and interaction between an airplane and a pilot. Complex simulation object interaction also can be shown in the same type of diagram, but it is not as easily designed and implemented in software.

The first type of interaction of objects addressed in this thesis is the interaction that takes place conditionally between two or more simulation objects. An example of this multiple object interaction is the collision between two separate vehicle objects.

The second type of interaction addressed is the interaction of moving objects with environmental objects such as terrain or weather. An example of this type of interaction is an aircraft colliding with terrain or terrain-following by a ground-based moving object.

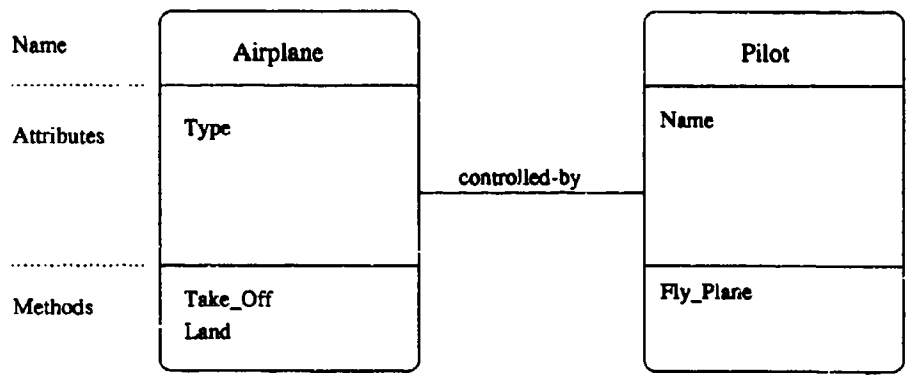

Figure 1. Object-oriented representation of an airplane and a pilot

The third type of interaction addressed is the interaction between parts of aggregate, or complex, objects. A complex object is an aggregation of smaller objects that interact to affect the behavior of the aggregate object as a whole. An example of a complex object in a battle simulation would be an aircraft that consists of a fuel subsystem, an engine subsystem, and other components the simulation modeler considers important enough to model. An example of the interaction at this level would be the fuel subsystem telling the engine subsystem that there is no more fuel.

Designing an object interaction model for use on a parallel computer introduces complications into the model that do not exist in sequential simulation. The designer must consider how the objects are to be distributed among the processors on the parallel computer. All aspects of object interaction must be considered to best balance the processing load of each parallel processor while trying to achieve a minimum level of communication between the processors. Minimization of communication between the processors is important because the speed of the communication channels between the processors is usually significantly slower that the computation speed of the processors themselves.

1.2.2 Battlesim. The Air Force Institute of Technology (AFIT) parallel battle simulation, Battlesim, divides a simulation battlefield evenly among parallel processors so that each processor is simulating one small part of the battleground. Figure 2 shows an example partitioning of a simulation into eight sections. Each object in the simulation is assigned to the processor that is simulating the part of the battlefield that corresponds to the object's current location. The main problem with this technique as it applies to this research is that there is no well-defined method of representing environment objects such as terrain or weather that have an effect on all the simulation partitions. Three possible approaches to implementing environment objects in a parallel simulation are to duplicate the entire environment object on each parallel processor, divide the environment object into smaller parts and put the appropriate part on each processor, or to represent it as single object on a single processor and require communications between the processors to determine if interaction exists. These three models are investigated in the course of this thesis.

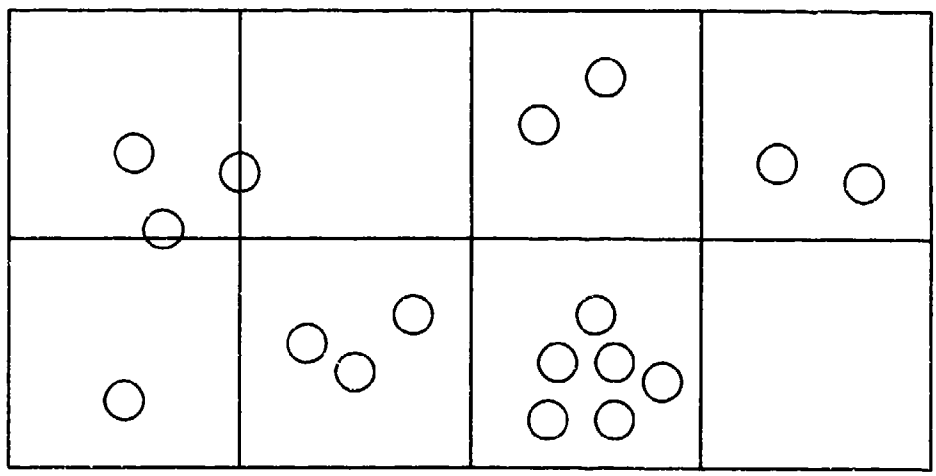

Figure 2. Example spatial partitioning of a simulation.

1.3 Problem

The current AFIT parallel battle simulation program, Battlesim, allows only simple interaction between simple objects such as the interaction between two aircraft. Currently, there is only one type of simulation object that is affected by a given set of object interactions. More complex interaction between objects is required in the program, as well as the ability to introduce other types of objects to the simulation. The types of interaction that are required include interaction between simple simulation objects, interaction between simple objects and environment objects, and interaction between the component parts of complex objects.

The purpose of this research is to address these three problems of object interaction and how their solutions can be applied to the Battlesim program. The first problem involves modelling the interaction between regular simulation objects. The second involves the problem of parallel simulation of environment objects and their interaction with the other objects in the simulation. Of specific concern is the interaction of vehicle objects with terrain, to include collision detection and terrain following. The third problem is that of parallel simulation of complex objects. The specific example examined is that of an aircraft, which consists of an aggregation of subsystems that interact and have an effect on the performance of the overall aircraft.

The goal of this thesis is to develop a model for use by the Joint Modelling and Simulation System (J-MASS) Program. The current simulation capabilities of J-MASS allow the simulation of radar transmissions, but plans exist to extend the capabilities of the simulation to allow the modeling of other applications.

1.4 Summary of Current Knowledge

The research efforts that participated in the evolution of the Battlesim program began in 1990 when Rizza developed a battle simulation program to simulate the battlefield environment (23). His research consisted of modeling objects whose interaction consisted of the detection and reaction to

collisions with other moving objects. The algorithm that he used involved computing the occurrence of a collision of an object in the simulation world by accessing a master object list and sequentially checking the position and velocity of every other object in the simulatio. . This work did not include parallel simulation of the battlefield.

In 1991 Moser developed a parallel simulation involving the collision of pool balls based on the 'Colliding Pucks' problem researched at Cal Tech (18). In order to break the problem down for parallel processing, he partitioned the simulation environment into strips of equal size. Instead of implementing a single master list of objects, he defined each sector to have its own object list that consisted of the pool ball objects and the section of the pool table that were located within that partition. To determine if there was a collision, the ball objects needed to be compared only against the other balls in the same sector and the edges of the pool table in that sector. This partitioning of the battlefield greatly reduced the search space of the problem of determining if a collision occurred because fewer comparisons needed to be made. Because the environment was partitioned, each ball object also needed to check its location in relation to the partition edges so that it could be passed to the next sector if it crossed a sector boundary .

Bergman's work in 1992 combined Rizza's battle simulation program with the partitioning technique used by Moser (2). This research led to a partitioning of the battlefield simulation program and converted it to a parallel, discrete-event simulation. The new battlefield simulator allowed each object to interact with other objects within its partition and to cross into a new partition if a boundary is reached. Additionally, a sensor capability was implemented to allow the detection of other objects from a distance, including objects that reside in adjacent sectors if they are within the sensor range.

This research effort continues the work of Bergman and makes the Battlesim simulation more robust by allowing the use of more complex objects and more complex object interaction.

1.5 *Scope*

The specific objective of this research is to develop and test a method that can be used to model complex interaction between battlefield objects. A general model that allows the modeling of all types of objects is analyzed, designed, and tested. The different types of allowable simulation objects are categorized and included in the model. Categorization is necessary due to the difference in the modeling of simple, complex, and environment objects.

The simulation model is designed to handle all types of battlefield objects. However, the final implementation consists of only the demonstration of the interaction of regular simulation objects by supporting multiple types of simulation objects and their object interactions. Environment objects that can be supported by the model but are not implemented are terrain, weather, and radio communications between the battlefield objects. Actual implementation of these objects is complex and beyond the scope of this thesis.

1.6 *Standards*

The object model designed adheres to the standards consistent with the principles of object-oriented simulation. The model ensures that objects used in the simulation are independent of other simulation objects and can be designed and implemented independently of the other objects in the simulation. Each object is designed using a 'black box' approach to the outside simulation world. Each object knows what data it needs for input and what data it outputs. The object interacts with the simulation support system for its input and output and is not aware of other objects in the simulation system.

Even though a true object-oriented language is not used, the implementation follows object-oriented programming principles of data encapsulation so that each object's information is contained in one location.

1.7 Approach

The first step in solving this problem was to investigate the current use of object-oriented programming and its use in simulation. This process consisted of a literature search and evaluation of previous AFIT thesis efforts. Among the topics researched were existing object models and their applicability to the battlefield simulation problem. One particular model evaluated was the current Software Structural Model (SSM) used by J-MASS.

Following the analysis stage, a design model was developed to model the representation of simulation objects and their interaction. This was be done by evaluating the specific requirements of the AFIT parallel simulation research group and applying the knowledge attained during the literature search.

Once the model was completed, it was implemented and tested using the Battlesim simulation program operating on a single processor. Implementing the model in a sequential simulation first allowed testing of the actual model without the complexities added by parallel execution.

After the implementation of the model worked in a sequential programming environment, it was modified for use in parallel execution. Conversion of the model for parallel execution did not involve significant changes since the problem was already mapped to multiple logical processors (LPs). Changes were required to keep objects synchronized between LPs. This model was then tested.

The implementation was complete when the model was operational and tested. At this point, the Battlesim program was able to simulate interaction of objects with other types of objects. The interaction of objects with terrain and the interaction of components in a complex object were not tested, but can be implemented using the new simulation model.

1.8 Outline of Thesis

Chapter 2 of this thesis contains background information in the research area resulting from a literature search of current research in the area of object-oriented simulation. Chapter 3 is a discussion of design issues that were considered before arriving at a final design method for interacting objects in simulation. Chapter 4 contains the actual design of the object interaction scheme resulting from this research. Chapter 5 contains information about the implementation of this scheme using Battlesim. Chapter 6 contains the results of the implementation and testing, conclusion about the results, and recommendations for further research in the topic.

Appendix A contains a data dictionary of the data elements of the design model and of the data elements of the Battlesim implementation model. Appendix B contains a description of the new Battlesim player structure. Appendix C is a user guide for using the model in the Battlesim program and modifying it to accommodate new player classes and event classes. Appendix D defines the new map object. Appendix E defines the new scenario file format. Appendix F contains configuration control information for Battlesim.

II. Background

2.1 Introduction

Current trends in simulation have led to the use of object-oriented techniques for modeling real-world objects and processes. The techniques used in Object-Oriented Simulation (OOS) are somewhat new to simulation and OOS languages are being developed to provide the capabilities of object-oriented programming for use in simulation.

The purpose of this chapter is to provide a background in the topic of object-oriented simulation and modeling as well as object interaction and collision detection. This literature review discusses discrete-event simulation, object-oriented simulation, the principles of object-oriented programming, object-oriented modeling, object interaction and the physical interaction of objects. The information for this chapter was drawn from current literature in the areas of discrete-event simulation, object-oriented simulation, collision detection, and Battlesim.

2.2 Discrete Event Simulation

Discrete event simulation is a simulation paradigm that advances in time increments that depend on the occurrence of events in the simulation and therefore has "the advantage of speed of program execution because events are scheduled only as needed (6:167)." Events are predicted based on the current simulation state and then saved until the simulation clock advances to that time. A discrete event simulation can proceed by either predicting and executing one event at a time or by predicting multiple events and scheduling them on a time-ordered queue and removing the earliest time event to be executed. In either of these methods, the simulation clock is always advanced to the time that an event is scheduled to occur before the event is executed (10).

2.3 *Object-Oriented Simulation*

Simulations provide a model of the behavior of objects or processes that occur in the real world. The use of simulation to model objects and processes allows modelers to obtain predictions of the actual behavior of an object or process without requiring implementation of the real thing. Simulation, therefore, can be a valuable cost saving tool and is used often during the design and analysis of future systems (3:195).

Early work in simulation focused on controlling the occurrence of simulation "events" in time. The simulation modeler identified the states and events of the system and then defined the conditions that caused a system to change state. This method of simulation modeling is good for simulating processes that have well-defined states, events, and state transitions. A newer generation of simulation methods emphasized the flow and processing of entities through a system. This method of simulation is good for modeling manufacturing processes (3:195).

The newest approach to simulation modeling is the object-oriented approach. This approach emphasizes the objects in a system and their interactions with each other. Object-oriented simulation (OOS) languages are good for simulation modeling, because, as Bischak states, "It is natural to view the real world as a set of objects that interact with each other" (3:194). It is also easy to map things that are not physical objects into simulation objects when modeling a system. An example is that of a database record which, though intangible, can be modeled as an object.

OOS languages are programming languages that use the object-oriented approach to modeling and provide tools that are necessary for use in simulation. OOS languages place emphasis on the objects in the system to be modelled while other types of simulation languages require the use of predefined simple objects that cannot be modified by the modeler. OOS languages allow modelers to create their own definition of new types of objects easily. An object-oriented simulation language is also concerned with communication between objects that results in the change of state of an

object. Conventional simulation languages, on the other hand, emphasize the use of function calls to which variables are sent to have their values changed (3:195).

Object-oriented simulation software is more reusable than traditional simulation software because objects can be reused in future simulations. Once an object is defined, its structure can be kept in an object library and reused later to define similar objects. Objects are also easy to maintain because all the information about the object is held in one place, within the object definition (3:195).

2.4 Object-Oriented Programming Principles

A simulation object consists of a data type that defines the object's attributes and the set of operations that can be performed on that data type. Data for the objects are encapsulated, which means that the data are accessible only through the operations that are defined on the object (3:194). The data are protected from being read or modified by other objects.

Descriptions of types of objects are called classes. A class is an object description that is a template for the creation of objects of that type. The class is described in terms of the types of data that are used to define objects of that class and the ways in which objects of that class interact with the outside world. Objects of a given class are called instances of that class. Object classes are stored in a library that can be used to write simulations (3:196) (27:303) (25).

Classes are defined in a class hierarchy, where some classes are subclasses of other classes. By making one object class a subclass of another object class, objects can inherit properties of other classes. Inheritance allows existing objects to be easily modified to create new objects. Objects can be retrieved from the object library and customized for various applications. For example, a *car* object class can inherit properties of a more general class called *vehicle*. This simplifies the definition of the *car* object class by not having to repeat the part of the definition that describes it

as a vehicle (3:196-197). Classes at a level in a class hierarchy are usable for defining other classes at the same level or any lower level (27:303).

Polymorphism is the ability to make different objects appear to do the same thing. One example is that of a *forklift_truck* object and a *stacker_crane* object. Each object could have the capability to pick up something using a function called *pick-up*. The program determines which function to call at run-time rather than at compile-time since the actual behavior of the function is determined only after it is known which object is involved in the function call. Polymorphism is also known as function overloading because the same function name is used in more than one object class (3:199-200) (17:329).

The use of dynamic objects requires tools to create and destroy objects on demand. Creation of an object involves execution of a constructor and must be done whenever a new object is required. Similarly, a destructor is executed to destroy an instance of an object and must be done when an object is no longer needed by the simulation. Use of the object creation and deletion processes allows the simulation to use computer memory more efficiently by having space allocated only for objects that are currently active in the simulation (3:199).

Simulation objects communicate by sending, receiving and interpreting messages. OOS processes are the property of the objects that contain them. When an object needs to interact with another object, it calls the required process in the other object to provoke a response. Because all of an object's data are encapsulated in the object, the only way the data can be accessed is by calling the appropriate process to make the object respond and perform the required action (3:200).

2.5 Object Modeling

The actual modeling of a system into a simulation requires an understanding of all of the objects and their interactions, since these are the key concepts in object-oriented modeling. Modeling of the objects themselves is a direct mapping of real-world objects to simulated objects. The

simulation modeler defines the objects by determining what objects to use, the types of information that they must store, and the operations they must perform. The modeler references the object library to obtain the types of objects needed for his or her application. The simulation model is then built using predefined-defined object classes and applying the principles of inheritance and sub-classes to create new and more complex objects for specific applications.

An example class hierarchy provided by a simulation language consists of base classes, application support classes, and application-specific classes. Base classes are at the top of the class hierarchy and are general-purpose object classes that are not necessarily unique to simulation. Members of base classes are often commonly used data structures such as stacks and queues. Application support object classes are created to be used in any simulation and can be reused in many application-specific problem areas. The simulation support object classes are members of the application support classes and may consist of simulation clocks and simulation output objects. The lowest level in this hierarchy consists of application-specific object classes that can be reused to solve problems within a specific application area. This level is where most simulation modeling must take place (27:303-304). The main goal of this class hierarchy is to create simulation classes that are simple and highly reusable, so they can be reused and expanded to allow more complex objects to be defined (27:304).

Sweeney provides an advanced look at object-oriented modeling. He defines a physical model that can be used to model actual physical objects. A physical model has mathematically defined physical properties and holds the state of an object at a given instance in time. The environment containing the physical model can have forces such as gravity, friction, and wind that act on the objects. The object models in this environment interact with each other and apply collision forces to keep objects from passing through each other. The physical properties of the objects determine whether objects bounce or break when they collide (31:1188).

2.6 *Object Interaction*

Object interaction is the method of "how each object is going to communicate with each other and the simulated world as well" (9:317). Objects communicate by passing messages. A message consists of a *receptor*, a *selector*, and possibly some extra arguments. The *receptor* indicates the object to which the message is addressed, the *selector* specifies which of the object's operations is selected, and the parameters may be needed for processing (17:328). For typical object interaction, object methods are used that emphasize object interaction rather than data manipulation. To call a method, only the identified method of the object needs to be called, along with the event time and event code (27:302-307).

One of the problems in modeling the interaction of objects is determining the visibility of objects to each other. In the following two models, objects do not know about other objects in the simulation. The objects communicate with other objects by sending and receiving data through local communication ports. The actual communication between the objects is handled by the simulation system.

2.6.1 Object-Connection-Update (OCU) Model. One technique used for modeling interactive objects is the Object-Connection-Update (OCU) Model (15). The OCU model divides the simulation domain into a collection of subsystems. Each subsystem may represent an object or related group of objects in the simulation and is defined by the following four parts: a controller, an import area, an export area, and the objects of the subsystem. Figure 3 shows how these areas may represent a group of objects.

The controller carries out the mission of the subsystem by managing the behavior of the objects in the subsystem and the communication between them. A controller is able to update, initialize, stabilize, configure, and destroy objects. A controller has no knowledge of the other subsystems in the application.

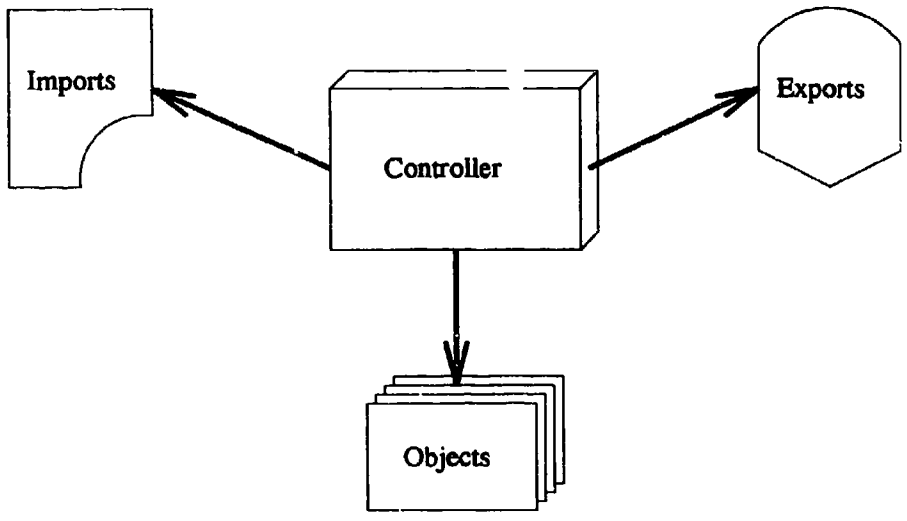

Figure 3. OCU model representation of an object.

The import area of each subsystem is the data input interface where each object retrieves the data that it needs. The import area retrieves the required information from the export area of other subsystems when the data is requested. The data in the import area is then either directly accessed by the objects or it is delivered to the object by the controller. The import area can be implemented as a "procedural interface for each kind of data needed from other subsystems."

The export area is an output interface through which data is output to other subsystems. Data placed in the export area are available to the import area of other subsystems. The export area can be implemented as a set of data records that is loaded by the local subsystem objects or the controller and is accessed by the import areas of other subsystems.

The objects in this model are passive and act based only on data received from other objects. They transform received input data into object state data and are not aware of where the data come from, only what data they need and that they can get it from the import area. The actions

of the objects are activated by the controller according to the mission coded in the subsystem controller (15:17-21).

2.6.2 J-MASS Software Structural Model. Like the OCU model, the J-MASS (Joint Modeling and Simulation System) Software Structural Model (SSM) divides the simulation objects into smaller systems. The SSM groups objects into teams, which consist of a set of objects and the operations that support them.

The objects are defined at three different levels: elements, assemblies, and players. An element is the lowest-level model component, which usually represents a physical component of a complex object, such as a tire on an aircraft. An assembly is an intermediate-level modeling component that represents an aggregation of lower-level components such as the landing gear on an aircraft. Assemblies may be a subcomponent of a higher-level modeling unit called a player. A player is the high-level component that represents a class of objects that exist as independent entities in the simulation, such as an aircraft. Figure 4 shows the relationship between the object components (1:22).

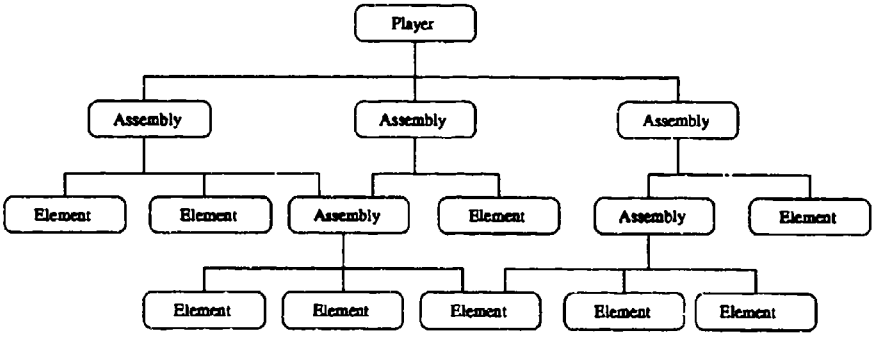

Figure 4. Object diagram for the J-MASS SSM model.

The SSM uses the Data Management Package (DMP) to manage all aspects of data access within the simulation. The DMP uses ports to allow objects to send and receive data from other objects. Ports are defined in the simulation model to be data channels between specific players. When a player wants to output data, it executes a DMP procedure call that sets up outputs in the port. When the port is ready, the other player removes the data from its end of the port. The use of ports allows the data exchanges to be done locally at an abstract level without each player needing to know the location and interfaces of the other players with which it must interact (1:20).

2.7 Physical Interaction of Objects

2.7.1 Maintaining Object Spatial Locations. When modeling battlefield simulation or other simulations that require physical interaction between moving objects, it is necessary to maintain the position of all of the objects. The two basic ways of handling position maintenance is to have the objects maintain their own position or to have an object manager that keeps track of all of the objects' locations. The simpler model is to allow the objects to maintain their own positions. The disadvantage of this method is that it results in a high number of communications between objects in order to find the spatial positions of all of the other objects.

Zeigler provides an example of an object manager by using what he calls a "space-manager" for a robot management system. The space-manager is used to keep track of where objects are located and with which other objects they can communicate and interact. When a robot object moves, the object's motion-system sends the new location and direction to the space-manager, which maintains this information. The space-manager also controls communication by passing messages only to other objects within a certain communication range. The space-manager is also used to detect collisions between the robots (32:238).

The J-MASS SSM uses a *Spatial Manager* to keep track of spatial information such as position, orientation, and velocity for all of the objects in the simulation. Players in the SSM are allowed

to update their state whenever it changes by providing their new spatial state or the change in their spatial state since the last update (1:20). The advantage of this type of model is that all of the spatial state information is centrally located. Additionally, the *Spatial Manager* can be used to provide services to the objects, such as providing the location of other objects or the spatial relationship between two objects.

2.7.2 Collision Detection. Collision detection and response provide realism to simulation because they prevent objects from moving through each other. Collision detection is the process of determining if two simulation objects are trying to occupy the same physical space. The procedure used to detect a collision depends on the type of simulation being used.

A discrete-event simulation requires that collisions be predicted as part of the process of determining the next event for an object. This is necessary because discrete-event simulation depends on predicted events and does not allow any real-time detection of object interaction. The process consists of comparing an object's position and velocity against the position and velocities of the other objects in the simulation to determine if a collision will occur.

Collision detection in a time-driven simulation consists of periodically comparing the positions of all of the objects to determine if two of them are occupying the same space. This is usually done during an object's position update cycle. An object is updated and then checked to see if a collision has occurred (34:4).

Response to a collision differs depending on the purpose of the simulation and how the simulation objects are modelled. A collision could result in objects bouncing off of each other, destruction of objects, partial damage to objects, or just notification that the collision occurred. The simulation modeler decides what the result will be. Response to a collision can be implemented by having the objects involved in a collision send each other a collision message that causes them to react to the collision. The collision message could contain such data as the identity or type of object and its weight, or any other items that could be used to calculate the result of a collision (29:66).

2.7.2.1 NPSNET. One example of collision detection is the NPSNET system at the Naval Postgraduate School. NPSNET is a real-time simulation program that participates in the SIMNET distributed simulation. NPSNET uses an algorithm that constantly checks whether a collision has occurred. As soon as a vehicle is updated, its position is updated and a check for a collision is made. The scope of the collision detection search process must be severely limited to maintain the speed of the simulation. Collision detection with fixed objects such as buildings or terrain is done only if the moving vehicle is below a threshold elevation. However, collision detection with other moving objects must be done at all times. If the X and Y position of the other vehicle is within a certain range, a second level comparison is made. This second-level check calculates the actual distance between the objects. If this distance is within the bounding spheres of the objects, then a collision has occurred (34:4).

2.8 Battlesim

The TCHSim program, used in the parallel simulation laboratory at AFIT, supports parallel discrete event simulation by providing a simulation driver, a simulation clock, a next event queue, and an interface to the SPECTRUM parallel environment (12) (21). The next event queue is a time-ordered priority queue that contains the set of events that have been scheduled. Each event has a time, an event type, and references to objects that are affected by the event (13).

Battlesim is a parallel discrete event simulation program written in C that is used to simulate the actions of objects as they move throughout a battlefield. The program is designed so that sectors of the battlefield can be assigned to different logical processors that can then be executed on separate processors of a parallel or distributed computer. This section discusses the state of Battlesim prior to this research effort (referred to herein as the "original" version).

2.8.1 *Limitations of the Original Version.*

2.8.1.1 Simulation Players. A simulation player in this application is an object that is being simulated. The original system was designed so that a single player class structure could be used to represent multiple types of players by using a player type identifier. Although this approach has worked well to support the player structures, it does require moving unneeded amounts of "data" between processors for simple players that do not use all of the generic attributes. The biggest limitation, however, is in implementing different versions of the operations for the different player types.

2.8.1.2 Event Prediction. The Battlesim event prediction and scheduling models assume that all types of simulation objects (player) are subject to the same types of events. The simulation, therefore, ran all the players through the same event prediction routines. This was not a problem as long as all the objects simulated were similar enough to have most of the same events. Exceptions, such as sensor detection events, were handled by checking first to see if the player had any sensors.

2.8.1.3 Object-Oriented Principles. The original implementation of Battlesim might be considered more object-based than object-oriented. Objects were not fully encapsulated and information hiding was not well enforced. Examples of these problems are listed below.

- The player objects had references to simulation sectors and player copies of themselves that existed in other simulation sectors.
- Methods for the player existed in files not specific to the player.
- Entities outside the player object directly accessed data within the player without using the proper function calls.

2.8.1.4 Implementation of Parallelism. The original Battlesim software did not have the capability to exchange data between multiple logical processors. This was due to the lack of a proper SPECTRUM interface for TCHSIM and the need for the proper SPECTRUM adjustments in Battlesim to ensure proper message passing, processor utilization, and processor termination.

2.9 Conclusion

This literature review provides a background in the areas of object oriented simulation and modeling. Object-Oriented Simulation (OOS) provides a natural way of modeling complex objects by providing a direct mapping between the real world and the simulation environment. This promising approach to simulation modeling allows for easier modeling and more reuse of simulation code.

III. Design Issues of Interacting Objects In Simulation

3.1 Introduction

The proper design of an object-oriented simulation requires addressing many related topics. Each of these topics must be reviewed in order to carefully plan and design the simulation system. This chapter discusses the issues involved in the representation of objects, communication between and within objects, keeping track of objects, and the problems introduced by parallelism in an object-oriented simulation. Each of these topics is discussed, as well as alternatives for their implementation.

3.2 Object Representation

Object representation is the method of defining an object in an object-oriented program. This section discusses the representation of general simulation objects, aggregate objects, and environmental objects.

3.2.1 General Object Model. One of the goals in designing a simulation environment is keeping the model general enough so that the environment can be used to simulate many types of objects. Although a given simulation system cannot simulate absolutely everything, it should be able to simulate all or most objects that can occur in a given class of simulation. In the case of this thesis, the simulation class is objects that move and interact with each other in a spatial environment.

Objects are typically represented as a set of attributes which comprise their state, and a set of functions or methods that allow outside entities to access, modify, or control their state. The design of a simulation system should allow all types of simulation objects to be accessed and handled the same way, using the same method calls for the same type of data access or notification of events. The object-oriented programming concepts of *classes*, *subclasses*, and *inheritance* support the ability of the simulation to handle objects the same way. The concept of *polymorphism* allows

the same calls to be used with different types of objects for activities or methods that perform the same function. One of the results of being able to treat objects the same and allowing the same calls to be used with different object types is that objects can be added to the simulation with minimal modification to the simulation environment and support system.

In an ideal object-oriented simulation environment, objects are completely self-contained, meaning that they encapsulate all of their attributes and control access to them with their methods. Objects are controlled by the simulation and by other objects through calls made to the methods of the controlled objects.

3.2.2 Aggregate Objects . Aggregate objects are simulation objects that are composed of component objects (17:342). For example, an aircraft may be represented not only as a single object known as the aircraft, but it may be modeled more complexly as an aggregation of objects that make up the aircraft and interact together to make the aircraft operate. These component objects could be a fuel system, engine system, communication system, navigation system, and other systems that may consist of the actual components that make up those systems. The amount of detail used to model these subsystems depends on what is intended to be simulated and studied. A radio communication system would not be very important in a simulation that is used to study the flight performance of a new type of aircraft, while the controls of the aircraft and their response would be very important.

Aggregate components can either be dependent or independent of the aggregate object. The existence of dependent components depends on the existence of the aggregate object. Independent components can exist without the aggregate object (17:343).

Aggregate components can also be defined as being shared or exclusive components. Shared components can be shared by more than one aggregate object, while exclusive components can be part of only one composite object (17:343).

A simulation model must be able to represent aggregate components in at least one of the following two ways. The components can either be represented as independent simulation objects that are tightly coupled, or they may be modeled as subparts to the simulation object that they comprise and be controlled by the "parent" object. Using the first method of having each component modeled as an individual simulation object, the components can be independent of the aggregate object. An example of independent aggregate components are planes that make up a squadron, where the squadron is an aggregate object. The second method of modeling components as subparts to a simulation model requires the components to be dependent on the aggregate object. The aggregate object is the only object that has visibility to the component objects. An example of this method is an aircraft and its subsystems.

3.2.3 Environmental Objects. In this thesis, an environmental object is an object that is considered to be part of the environment where the simulation is taking place and affects objects at any location in the simulation. The objects are not typical simulation players because they often cover every partition in the simulation. Some examples of environmental objects are terrain, weather, atmosphere, electro-magnetic field, or any other object that can affect players in a simulation.

Environment objects can be implemented in one of three ways on a partitioned simulation. The object can be totally duplicated in each partition, partially duplicated in each partition, or solely existent on one partition. The method chosen for design and implementation of the environment object depends on the characteristics of the object. Objects such as terrain require a different set of complex data for each partition and are too large to implement fully on each partition. Therefore, it is more efficient to implement a partial terrain object on each partition, containing only the terrain data necessary for that partition. Since the goal of using spatial partitioning is to minimize the search space of interaction calculations and to minimize communication between

processors, implementing an environment object on only one partition is a poor design choice if many of the simulation objects must interact with it.

The desire to keep the simulation easy to modify makes it more desirable to design environmental objects like any other simulation object. This would alleviate the need to provide special object handlers for environment type objects.

3.2.3.1 Representation of Terrain. There are two major methods used to represent terrain in a simulation. It can be represented as a system of map coordinates with respective altitudes referenced for each coordinate, or it can be represented as a set of polygons which represent pieces of the terrain. Sometimes these two methods are mixed so that points from map data are referenced together to form polygons.

Terrain as Map Data. Terrain is most easily maintained as map data because it is simply a table of map coordinates where each coordinate has an associated altitude. The Defense Mapping Agency (DMA) is a common source of map data. The DMA provides Digital Terrain Elevation Data (DTED) that is arranged in a rectangular grid and provides elevation data for the locations corresponding to the intersection of the rows and columns of the grid (24) (8:2-1).

The advantage of representing terrain as map data is that altitudes can be easily referenced by using the map coordinates. The data can be sorted in a two dimensional array. The disadvantage is that it is simply point data and requires translation to a new format to be able to graphically display it and interpolation to determine altitudes in locations that are not at the specific points used in the map dataset.

Terrain as a Set of Polygons. Representing map data as a set of polygons is common when the data must be displayed graphically, since graphic display systems are designed to handle the display of polygons. Polygon data is usually represented by a set of vertices, a normal,

color, and a texture. The normal, color, and texture are used to affect the appearance of the graphical representation of the polygons.

A polygonal model of terrain can be generated from map data by joining the corners of each grid cell to form a non-planar rectangle and then connecting two opposite corners of the rectangle to form two three-sided polygons. Using every gridpoint in map data to generate a polygonal terrain file often results in terrain files that take up substantial memory and contain more detail than necessary. The precision and storage requirements of the resulting polygon model can be lowered by using only every nth grid point or by choosing a representative data elevation for each area of a larger, or more coarse, grid (8:2-2).

Representation of terrain as a set of polygons eliminates the need to calculate pieces of terrain information for display or when checking for collision detection. The algorithm must simply check for intersection between an object and an existing polygon. However, it introduces problems that are not encountered when using the regular map data representation. The problem of increased data storage occurs since vertices are referenced more than once. The problem of referencing the correct polygons occurs since the data must be stored differently than map data. The map method allowed data points to be referenced directly, since data could be represented as arrays referenced by X and Y locations, to find the altitude value Z.

3.2.3.2 Representation of Weather. Weather is an environmental object that can take the form of wind, clouds, rain, visibility or other weather conditions that can affect an object. In simulation it is often represented as a parameter that fits into an equation affecting the performance of an object. It is usually controlled by calculating a probability that controls the state of the weather.

The Saber system at AFIT represents weather as a set of states. The simulation is initialized with a forecast percentage. Weather for each section in the simulation is determined by computing the result of a random number with a forecast percentage index. The index determines the state of

the weather for that particular "weather period". Weather is updated at intervals that are multiples of simulation time periods (26).

3.3 Object Interaction

Object interaction in a simulation is the process of objects communicating with one another, either directly or indirectly, and possibly changing state as a result of the communication. This communication can be direct, as in an object sending a message directly to another object or indirect, having the simulation handle interaction that occurs due to the given states of the objects. An example of direct communication would be a supervisor object giving orders to a subordinate object, or an aircraft object sending a radio message to another aircraft object. An example of indirect communication is a collision between two moving objects that is handled by the simulation system. In this case, neither object "knows" about the existence of the other object or sends any messages. Communication between objects can be categorized into three different areas: to request a service , to provide notification that a specific event has occurred, or to provide data (19).

3.3.1 Object Visibility and Independence. Visibility is the term used to describe whether one object has knowledge of the existence and location of another object so that it can send it a message or call one of its methods. Visibility not only determines which objects are "seen" by an object, but it also determines what object methods inside an object can be seen by other objects. This section is mainly about objects being able to "see" other objects. To establish visibility of an object, the modeler must determine what objects need to communicate and what object data need to remain hidden from other object.

3.3.1.1 Objects With Visibility. When objects have visibility of other objects in the simulation, they are able to make calls to the methods of those objects. This is the approach most commonly used in object-oriented systems as described in object-oriented texts (25) (4). Normally objects are given visibility to other objects when they are active objects that control or directly

interact with the other objects. The controller, or calling objects, have visibility of objects that they call.

Common approaches used to give objects visibility to other objects are through inheritance, making one object an attribute of another, making a pointer to one object an attribute of another, including a header file that defines another object's class, and passing an object in a parameter to methods that require that object(28:23).

The disadvantage of giving objects visibility to other objects is the lack of flexibility in changing the system. When object types are added to or deleted from a simulation, the other objects that specifically access those objects must be updated to accommodate the changes.

3.3.1.2 Objects Without Visibility. When objects are not given visibility to other objects in the system, they cannot access them without interacting with a control object that can provide them a pointer to the object. Objects that do not have visibility to other objects are normally passive and are acted upon by other objects. These objects do not need to know about the existence of any of the other objects in the simulation. They only need the methods that allow them to react to inputs from other objects in the simulation, to perform their own behavior, and to provide and receive data as a result of control external to the object. If each of these objects is self-sufficient and unaware of other objects in the simulation, the matter of adding an object to a simulation should be a simple effort of plugging in a new object type, the event types that it takes part in, and the prediction algorithms for these event types, without creating changes all across the simulation.

The interfaces between each type of object may need to be defined, though, if those objects need to interact. For example, the simulation must know which objects can interact with each other. Since the objects do not know other objects are out there, the simulation must control their interactions by predicting events and then handling the interaction that occurs due to the event.

Therefore, each object must know how to react to a given event in the same way. For example, during a collision, an object is told that it was hit by a given force in a given direction, and the object must react accordingly. Unless that object has some kind of sensors, it does not know what type of object hit it. The object can be designed so that it reacts differently to different levels of impact. A small impact may cause a dent, or a change of direction. A larger impact may cause the loss of performance capability, and an even larger impact may cause total destruction of the object.

3.3.2 Object-Object Interaction. Simulation objects are simple objects that are being simulated. An example of a simulation object is an aircraft that has attributes and methods, but no component parts. Interaction results from calling one of the object's methods and the object responding to the call. If the objects have visibility to the other simulation objects, they can communicate by calling the object's methods directly.

If the objects do not not have visibility to the other objects, interaction must be taken care of indirectly by the simulation. The simulation analyzes the state of the objects and causes interaction by calling the methods of the objects that interact. Although some simulation scenarios contain simulation events that cause multiple objects to interact at the same time, at any one instant an object can communicate with only one other object by calling one of its methods.

3.3.3 Object-Environment Interaction. As described in an earlier section, environment objects are objects that represent a single object that is part of the simulation environment. This section will discuss the issues of objects interacting with environment objects and whether this interaction is different than the interaction between two simulation objects. Environment objects are omnipresent in the simulation and must consistently interact with the other simulation objects. If the environment objects can be modeled the same way as simple simulation objects, then their interaction can also be modelled in the same way.

3.3.3.1 Object-Terrain Interaction. The primary interactions that terrain has with other simulation objects is collision detection with the terrain for airborne objects and terrain following for ground-based objects. Usually terrain is a completely passive unchanging object that reacts physically with the simulation objects.

Collision Detection for Airborne Vehicular Objects with Terrain. If map data is being used to represent terrain, simply using the vehicle's position and referencing the respective map location will allow finding the altitude at the current X, Y location to see if the object has collided with the terrain. The actual collision detection between the vehicle and terrain must be determined by finding if the body of the object has intersected the plane of the local piece of terrain. Since map data is typically point data, the actual altitude and plane of the local piece of terrain must be calculated by using the corner points of the local piece of terrain. These computations are relatively easy to do. Other considerations here are when an object is at the intersection of two or more pieces of terrain. Computations must first determine which pieces of terrain are involved, then determine the actual coordinates of each piece, and finally determine whether an intersection has occurred.

Terrain Following for Ground Vehicle Objects. Terrain following for ground vehicles requires the same type of computations to determine the pieces of terrain involved, but a different type of computation to determine how the object will be affected. The slope of the terrain determines the orientation and roll of the vehicle as it travels across the terrain. Changes in the vehicle's state are represented by events that must be computed to coincide with the vehicle reaching places where the slope of the terrain changes. Determination of the change in slope will be affected by several details. The most important of these is the incremental size of the terrain. If the terrain is broken down into segments that are considerably larger than the object, say every 50m, then it is trivial to schedule an event every time the ground object intersects the edge of a plane of the terrain. However, as the terrain gets broken into smaller, more exact pieces, the object

must interact more and more with the terrain. If the terrain is measured in smaller and smaller pieces, it gets so that each wheel of a vehicle must interact with a different piece of the terrain.

The solution of this problem depends greatly on the goal of the simulation. If the simulation is just simulating the interaction of battlefield vehicles where terrain is not part of the calculation, it is pointless to perform calculations for every bump in the road. On the other hand, if the goal of the simulation is to perform a performance test simulation for a particular vehicle in rough terrain. every bump in the road can be very significant.

3.3.3.2 Object-Weather Interaction. Simulation objects interact with weather by calling the weather object's methods to determine the state of the weather. The state of the weather has an affect on the performance of the object.

3.3.4 Aggregate Component Interaction. An aggregate object has internal interactions and external interactions. The internal interactions are between its internal components that determine the behavior of the object. The behavior is not seen by the environment surrounding the object, only the resulting actions or changes in behavior or state that the object makes due these interactions.

Aggregate objects must have their components interact, either internal to the player or as individual objects. The simulation model used here assumes that all aggregate object communication is taken care of by the object, or that the individual components be treated as simulation objects that equally use the next event queue.

3.4 Event Handling

Events are used to represent the occurrence of interaction between objects or the change of state of an object. The subject of event handling in discrete-event simulation concerns the prediction of future events and then executing or reacting to the events when they occurs.

3.4.1 Predicting Events. Predicting events is done by examining the current state of the system and determining what future events will take place. The progress of the simulation depends on being able to determine the next possible event that will take place. This process is most successful when one event is predicted at a time, since the execution of an event can change the state of the simulation and cause any other already predicted events to become invalid.

Events often represent the interaction of objects and require knowledge of other objects in the system. For this reason, the responsibility of predicting an event needs to be given to objects in the simulation that have visibility to be able to access the appropriate objects.

3.4.1.1 Predicting Events for Aggregate Objects. In discrete event simulation, the modeler must decide how to handle event prediction and interaction with the next event queue. Communication with the next event queue may either be through the main object or directly from the component object. If the main object handles the scheduling of internal events, then all internal event interactions must be passed through the main object. On the other hand, if the lower objects handle their own interactions with the next event queue, additional intelligence must exist in these lower objects to be able to interact with the queue.

3.4.2 Reacting to an Event. Reacting to an event involves performing the actions required for the event to be executed. When an event is removed from the next event queue, it is executed. The execution of the event causes manipulation of the appropriate objects involved in the event.

3.4.2.1 Single-Object Events. The method of handling events that handle single objects is very easy. Single-object events are usually internal to objects and the events simply execute the proper simulation object methods to handle the event. Events that interact with other objects in the simulation are handled by the event manager.

3.4.2.2 Multiple-Object Event. Events with multiple objects need to be handled by an event manager to avoid excessive complexity within the simulation objects. The scheduled event must contain references to all of the involved objects and then provide the interaction control between the objects. This interaction control performs the functions of retrieving object state, modifying their state, and calling the methods that are required to produce the effects of the execution of the event.

3.5 Managing Objects

As addressed in the literature review, the position and movement of objects can be maintained and provided by a spatial manager or by the individual objects. Both of these methods have their advantages and disadvantages.

3.5.1 Spatial Manager. A spatial manager, such as the one used in J-MASS, keeps track of the location of all the objects in the simulation. The spatial manager provides an easy to use interface that provides such services as the location of objects, the nearest object, collision detection, and object visibility. Objects query the spatial manager to determine which is the next object that they will collide with. They know only the existence of the spatial manager, but not the existence or locations of the other objects in the simulation. The advantage of this approach is that each player only needs to know the existence of the spatial manager and how to interact with it. The spatial manager, however, needs to be complex enough to be able to determine the next collision for any selected object.

The spatial manager needs to be updated by the objects as they change. The two ways to handle these updates is either to get a positional update or just a velocity update. In the first case, the objects know their actual location in the simulation. In the latter, they only know their current speed and direction and the spatial manager determines their actual positions, as calculated by the changes in their course and speed.

The advantage of using a spatial manager is having all of the position information in one location and possibly even ordered to make positional searches more efficient. The disadvantage of using a spatial manager is that duplicate information is maintained in the objects and the spatial manager. Also, the spatial manager is likely to be difficult to modify when the simulation changes to require the use of different object types and the provision of different services by the spatial manager.

3.5.2 Object Self-Management. Having objects keep track of their own position keeps the object's information encapsulated within the object. This provides for a simpler design because object data does not need to be replicated in a spatial manager every time the object changes. All references to the object's location must be made by using the object's method to retrieve the data. The advantage to this technique is that object positional data is kept in one place and therefore does not require constant updates to a spatial manager. The disadvantage is that each object must be accessed individually to determine their status or the status of the simulation environment as a whole.

3.6 Impact of Parallelism

3.6.1 Logical Processor Synchronization. Executing a simulation on more than one processor requires synchronization between the processors to maintain the integrity of the simulation. In a real-time simulation, this process consists of keeping the simulation clocks on each processor synchronized. In a discrete event simulation, this process requires the simulation on each logical processor (LP) to wait until all of the other LP simulations have advanced to a time equal to or greater than the local simulation clock. Synchronization must be maintained so that simulation events across the multiple processors happen in the correct time order.

3.6.1.1 Conservative Approaches. One method of synchronization is the Chandy & Misra approach of not allowing a processor to proceed until all its input channels have reached a

time greater or equal to the time of the local simulation clock (5). This is considered a conservative approach. In a parallel discrete event simulation, this means not taking an event off the next event queue until all the input channels have reached a time equal to or greater than the time of the first event on the next event queue.

3.6.1.2 Optimistic Approaches. Time-Warp and Rollback are optimistic approaches to processor synchronization. They allow the simulation on each processor to proceed at its own speed and then back up if a message is received with a time-stamp earlier than the current simulation time. These implementations require a method to save the state of the simulation at regular intervals so that it can be restored if needed. Depending on the characteristics of the simulation these approaches are used in, they can also add a lot of overhead if the simulation must be backed up and restored often (14) (22).

3.6.2 Simulation Object Synchronization. In a simulation that is divided into more than one simulation sector, it is possible for an object to require visibility into other simulation partitions due to its proximity to the boundary of its current sector. This visibility is necessary so that interaction can take place with objects that are in other partitions, but are able to interact with an object because they are located close enough to the object.

3.6.2.1 Simulation Objects. Simulation objects moving between LPs require the addition and deletion of player copies from LPs as the objects move from one LP to another. However, when an object is located right near a boundary, there is more than one way to implement its visibility requirements. The object can either be duplicated on both of the LPs or it may just be given visi into the other LP by communicating with it.

Multiple Player Copies. In this case, the simulation places a duplicate player copy in the sector that now holds part of the player. The player copy is responsible for predicting and executing ents that occur in the second sector and cannot be detected by the original player

copy. Additionally, it is possible that the occurrence of an event on one processor may nullify an event that was predicted on a different processor.

When copies of the same object exist on more than one LP, the data in each of the copies must be consistent to maintain the integrity of the simulation. A process must be devised to maintain the consistency of the object copies. A method of keeping track of the other player copies must be used so that the copies can be updated.

Increased Object Visibility. One way to provide increased visibility is to allow an object to send messages to other LPs and have the other LPs respond if a certain object is found on one of the other LPs. This method requires more communication between LPs, since communication must take place between LPs every time an event is predicted for an object that is required to have visibility past a partition boundary.

3.6.3 Event Scheduling. When objects have multiple copies, a determination must be made as to how events are scheduled. If the main object copy does all of the event predictions, then it makes sense to schedule events for that object on the same LP as that object copy. But if a subordinate object copy predicts an event, the event can either be scheduled on the same LP as that copy, or sent to the LP of the main object copy.

Scheduling Events on the Local LP. The advantage of the first approach is that the event operates on the object copy that predicted it. This eliminates excessive communication of events back to the LP of the main object copy, and it allows the event to execute using the same information and objects that were available when the event was predicted. This allows the execution of the event to take place on one LP. The disadvantage is that multiple object copies are being modified on different LPs and require synchronization to ensure integrity of data.

Scheduling Events on the Owner Copy's LP. The advantage of sending a predicted object back to the main object copy is that all object modification takes place on the main object copy. The disadvantage is that the LP of the main copy may not contain all of the data necessary to execute the event. Communication between LPs may be necessary to accomplish execution of the event using this method.

Sending events back to the LP of the owner copy can be accomplished in two ways. The event can either be sent immediately to the other LP or it can be scheduled locally and then sent to the other LP when it is time for the event to execute. The second method is better for the use of parallel simulation communications protocol because it is consistent with keeping messages between processors in monotonic increasing order.

3.6.3.1 Environmental Objects. Environmental objects must either be maintained on all of the simulation LPs in which they reside or exist on only one LP and require communication between LPs in order to communicate with it. If the object is maintained on multiple LPs, it can either have full object copies on each LP or just have partial copies that are associated with that LP that the copy is located on. The two primary concerns of the representation of these items in a parallel discrete event simulation model are whether the objects can fit into the simulation model like any other simulation player object and how to model them across multiple sectors and processors.

3.6.4 Partitioning the Simulation. The simulation can be partitioned either by dividing the objects or by spatially partitioning the environment. The best approach depends on the object interaction patterns of the application.

3.6.4.1 Object Partitioning. Partitioning the simulation by dividing the object types has the advantage of keeping all like objects on the same LP, making them easy to find. This model would not require objects moving between LPs or the maintenance of multiple player copies.

The disadvantage of this approach is that it will require increased communication between LPs to detect and resolve interactions between objects that reside on different LPs.

3.6.4.2 Spatial Partitioning. Partitioning the simulation spatially is a good approach if objects interact with other objects in their proximity. This approach limits data search by requiring the search to consist only of objects in the local area. The disadvantage of this approach is the requirement to create and delete object copies as they move between the partitions and the maintenance of multiple object copies.

Partitioning of Environmental Objects. Environmental objects such as terrain must be represented on every partition of the simulation environment to avoid excessive communication between processors. This can be accomplished either by representing the environmental object in entirety on each partition or by representing only the portion of the environmental object that corresponds directly with the partition. The decision of which method to implement will depend on the characteristics of the environmental object, such as storage requirements.

One example that can be used to represent terrain files in partitions is that of the NPSNET at the Naval Postgraduate School. The NPSNET uses terrain files that each consist of one square kilometer of terrain data. NPSNET uses this method to page terrain in and out of the simulation as a simulation object moves through its environment (33:66). However, any simulation could use this method of terrain representation to select portions of terrain that apply to a particular partition of the simulation environment. Selectively representing terrain in this manner reduces storage requirements of the current terrain representation file and reduces the search space for collision detection.

3.6.5 Transparent Parallelism. Communicating with another object requires visibility of the location of the other object since its location must be known to ensure the message is sent to the right place. One practice that greatly simplifies the implementation of simulation is to

make communication with remote objects transparent. This allows communication to take place as usual, without the sending object "knowing" if the receiving object is local or remote. The aspects of finding the other object are handled by the simulation system (11:55). A common way to implement transparent parallelism is through the use of object request brokers, which support external message routing between objects that reside on separate processors (20:34).

3.7 Conclusion

This chapter provided a discussion of the topics considered in the design of the simulation model. Chapter IV discusses the design of the model and the design decisions that influenced its design.

IV. Design

4.1 Introduction

This chapter discusses the design of the object oriented simulation model resulting from this research. First the simulation model is defined. The design of the model is described, its major components are identified, and the execution of the model is discussed. Then a discussion is provided of how to change the model to apply it to different simulation applications. Finally, the design choices are discussed, as related to the discussion in Chapter III. This design is for a spatially partitioned parallel discrete-event simulation.

4.2 Description of the Model

The model for this Simulation Support system was designed using Rumbaugh's object oriented design and analysis methods (25). The first step involved identifying all of the objects in the system and the relationships between them. Figure 5 shows the resulting object diagram.

The model consists of the following objects: an Event Scheduler, Event Predictors, Events, Players, Partitions, Player Containers, an Object Copy Manager and a Relationship Map. Notice that both the Application Support System object and the Application object have Events and Event Predictors as components. The Relationship Map object is used to map all of the relationships between objects that are implied by dashed line object association connections in the diagram.

The Application Support System is an aggregate of objects that support a simulation. These objects take care of all Event handling, maintaining pointers to the Players, handling updates of Player copies, and maintaining association links between objects. The Application Support System also has a set of Events that are used for simulation maintenance, regardless of the application. These Events are used to maintain consistency between object copies among different partitions.

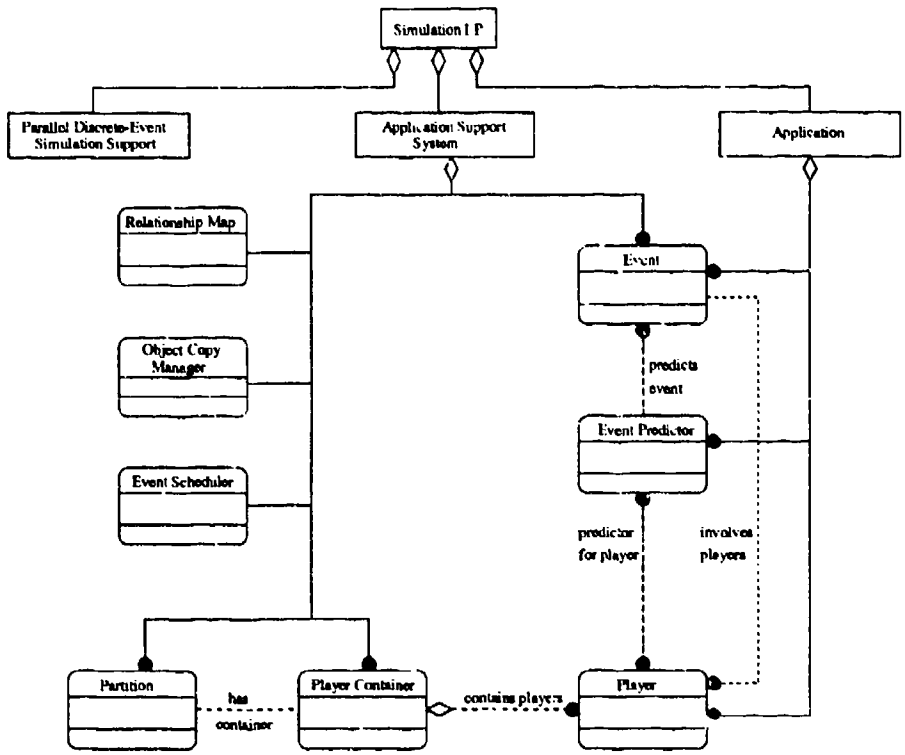

Figure 5. Object Diagram for the Simulation Application Support System.

The Application is an aggregate of the objects that are specific to a simulation application. These objects are Players, Events, and Event Predictors. This is the part of the model that will change as an application changes.

4.3 Major Components of the Model

4.3.1 The Player Object. A Player object is an object that is being simulated by the simulation. It can be stationary or mobile, simple or aggregate, positional or environmental. Based on the type of object, the Player object may or may not have Events scheduled. Figure 6 shows the object diagram for the Player object.

The Player object must have an object type, an object id, and its current state time as attributes so that it can be handled by the simulation. Other attributes that are required depend on what type of object it is.

All Player objects are required to provide object creation and destruction methods and methods to access their attributes. They must also have the ability to save their state so that the simulation can be rolled back to a previous time and restore the state of the simulation at that time. The Player object must also have methods that allow it to react to Events. For example, an object that is involved in a collision should have a method called *in_collision* that gets passed all of the information that the Player needs to handle the collision as it affects the Player. Necessary information includes items such as the mass and velocity of the other object or the direction and force of a collision vector that has hit the Player.

4.3.2 The Event Scheduler Object. The Event Scheduler object is an object-oriented representation of the object that schedules the next event for a Player object. The Event Scheduler determines the next Event for a Player object by first calling the Player to determine its next internally scheduled event. Then the scheduler queries each Event Predictor object that is associated with the Player object to return what they predict as the next Event for the Player object.

Player Object Diagram

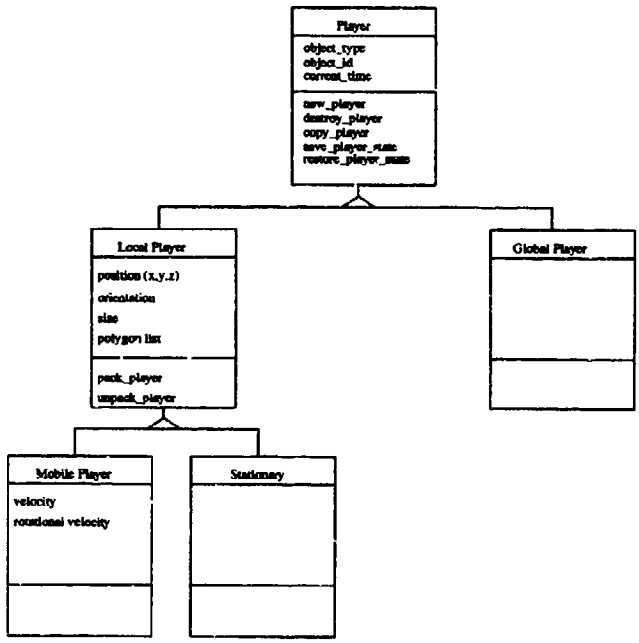

Figure 6. Object Diagram for the Player object.

The Relationship Map is used to determine which Event Predictors to use. The Event Scheduler determines the next Event by comparing the event times of the Events returned by the Player and the Event Predictors. The Event with the earliest time is then scheduled on the simulation event queue. It is possible that no Events are scheduled for an object as a result of this process, due to the current state of the Player objects.

4.3.3 The Predictor Object. A Predictor object is an object that predicts a specific type of Event for a Player object. Some Predictors may predict Events for a group of closely related Event types. The Predictor is queried by the Event Scheduler object to predict the next Event of the Event type that the Predictor is responsible for. The Predictor iterates through each of the Player objects in the Player Container for the Partition containing the Player having an Event predicted. The object type, or Player class, of each Player is compared against the Relationship Map to determine if that Player can participate in the interaction that that Event Predictor predicts. If the Player class is correct, the state of the Player is compared against the state of the Player having an Event predicted to determine if and when an interaction Event will occur. The Event Predictor returns to the scheduler the earliest time Event that it predicts.

4.3.4 The Event Object. The Event object is an object that represents an object interaction Event that has been predicted to occur in the future. The Event is created by a Predictor object, then placed on the next event queue to wait for its time to execute. Two different types of Events are simulation application Events and simulation support Events. Simulation application Events represent actual Players or state changes of Players that have been predicted to occur. Simulation support Events are used to initialize Player objects, maintain Player object copies, and maintain execution of the simulation.

The Event object has attributes for the Event time, Event type, and references to the Players that are involved in the Event. All Events must have an *Execute_Event* method that handles the execution of the Event. Figure 7 shows the object diagram for the Event object.

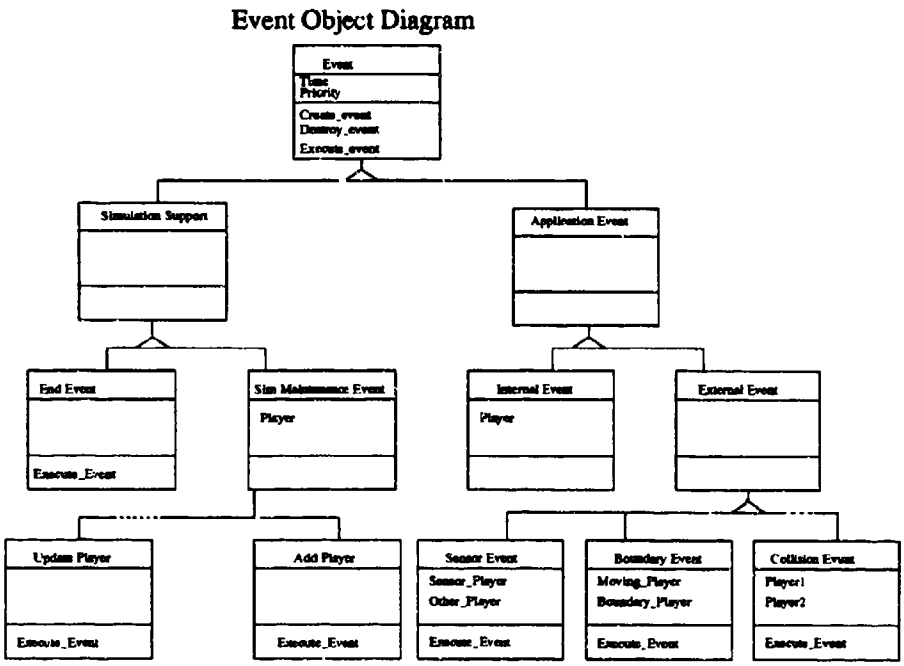

Figure 7. Object Diagram for the Event object

4.3.5 The Partition Object. The Partition object is a spatial object that represents a three-dimensional area of the simulation environment. Each Partition object is associated with a Player Container object that consists of the set of Player objects that exist in that partition. The Partition object has attributes defining the limits of each Partition and methods for accessing these attributes.

4.3.6 The Player Container Object. The Player Container object is a container object that contains a set of Player objects. Each Player Container object is associated with one Partition. A single Player Container object contains all the simulation Player objects that are in a particular simulation partition.

4.3.7 The Object Copy Manager. The Object Copy Manager handles the updating of remote object copies by sending object updates to the LPs where remote object copies reside. This is done by checking the Relationship Map object to find the appropriate partitions where the copies are located and then again to find the LPs where the partitions are located.

4.3.8 The Relationship Map Object. The Relationship Map object is a group of integer to set of integer maps that are used to map the relationships between objects. The maps can either be static or dynamic and can be used to model any relationship between objects in the simulation that can represent both items being mapped as integers. The modeler must set up a map and have initialize it. The map can then be accessed and modified during the simulation. The basic mappings that are required for operation of this model are listed below.

Player Class to Predictor Class This mapping maps each Player class to the set of Predictor objects that it uses to predict the next Event for a Player from that class. This map is static and identical on all of the LPs. Table 1 shows the relationship between Player classes and Event Predictor classes.

Predictor Class	Aircraft	Tank	Truck	Terrain
Collision Event	X	X	X	
Sensor Contact	X	X		
Enter Sensor Range	X	X	X	
Boundary Event	X	X	X	
Route Event	X	X	X	
Ground Course Change		X	X	

Table 1. Example Mapping of Player Class to Predictor Class

Predictor Class	Aircraft	Tank	Truck	Terrain
Collision Event	X	X	X	X
Sensor Contact	X	X	X	
Enter Sensor Range	X	X	X	
Boundary Event				
Route Event				
Ground Course Change				

Table 2. Example Mapping of Predictor Class to Player Class

Predictor Class to Player Class This mapping maps each Predictor class to the Player classes to use from the Player Container object while predicting an Event. This map is static and identical on all of the LPs. Table 2 shows the relationships between Event Predictor classes and Player classes.

Player Copy to Partition This map keeps track of which partition a particular Player copy is in. This mapping is used so that the actual Player copy doesn't need to know about partition objects. This map is dynamic and changes as Player copies are created and deleted from partitions.

Player to Partition of Owner Copy This map keeps track of which partition contains the owner Player copy of a particular object. This map is dynamic and changes as a Player moves between partitions. This map is identical on all LPs that contain a Player copy of the object.

Player Owner to Partitions of Player Copies This map keeps track of the partitions that contain a Player copy of an object, including the partition that contains the owner. This map is dynamic and changes as the object moves and crosses partition boundaries. This map exists only on the LP that contains the partition where the owner Player copy is located.

4.4 Operation of the Model

4.4.1 Event Handling.

4.4.1.1 Event Scheduling. The Event Scheduler object is used to determine the next Event for a Player. The Event Scheduler object first queries the Player object to determine its own next internal Event. Internal Events involve internal components that are not visible outside the object. On receipt of the internal Event, the Event Scheduler queries the Event Predictor objects that are applicable to the Player object. This relationship between the Player type and the Event Predictor type is established and referenced by using the Relationship Map object to map the Player object to the Event Predictor objects that schedule Events for it. Figure 8 shows the state diagram for the scheduler object.

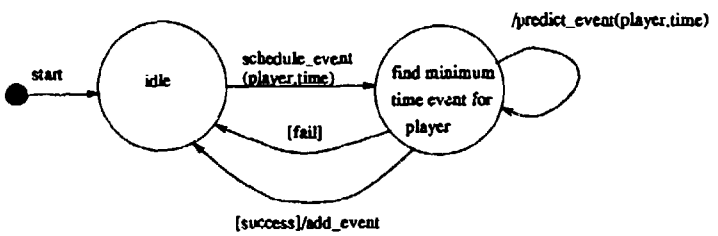

Figure 8. State Diagram for the Event Scheduler object

Each Event Predictor object in turn determines the next occurring Event for the Player involved and returns this Event, or NULL if none is predicted, to the scheduler. The scheduler

determines which of the Events has the lowest time and schedules this Event on the next event queue. It is possible that no Events are scheduled for a Player after this sequence. Figure 9 shows the state diagram for the Event Predictor object.

An example of this event scheduling process proceeds as follows. An Aircraft Player object requires its next event to be scheduled. The Event Scheduler queries the Player's method for determining its next internal event. The Player object determines its next internal interaction event, which in this case is a Route Point Event at time 25. The Scheduler then queries each of the Event Predictors that correspond with the Aircraft Player. The Collision Event Predictor predicts that a Collision Event will occur at time 23. This event preempts the Route Point Event since it will occur at an earlier time. The Boundary Event Predictor predicts a Front End Object Event will occur at time 29 to signify that the front of the Player has reached the edge of its current Partition. This Event will be discarded since it will occur after the Collision Event. If there are no more Event Predictors to query, the Collision Event is scheduled on the next event queue.

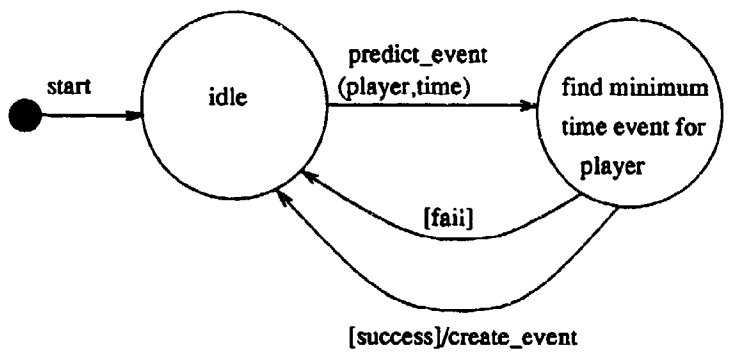

Figure 9. State Diagram for the Event Predictor object

4.4.1.2 Event Execution. When an Event is taken off the next event queue, its *execute_event* method is executed. Each Event type has its own *execute_event* method. The exe-

cute_event method handles all the object interaction that occurs as a result of the Event. Figure 10 shows the state diagram for an Event object.

An example of the execution of a Collision Event proceeds as follows. The Collision Event is taken off the event queue and its *execute_event* method is executed. The Collision Event first updates the states of the Player objects involved in the collision. Then it obtains the state of both Players involved in the collision by using the attribute access methods of the Players. Then the *do_collision* method is executed on each Player object. The *do_collision* method requires parameters such as mass, speed, and direction of the other player so that it knows how to react to the collision. Once the Player objects are modified, control returns back to the Event. The copies of the Players on remote partitions are sent updates by using the Object Copy Manager. Then already scheduled Events on the event queue involving the modified Player objects must be rescheduled to guarantee their validity now that the states of those Players has changed. Finally, the next events for the modified Players are determined.

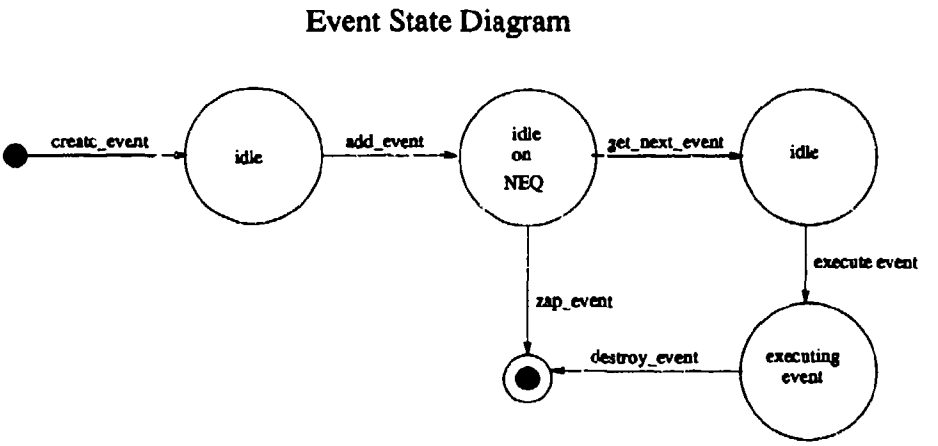

Figure 10. State Diagram for the Event object

4.4.2 *Object Interaction.*

4.4.2.1 *External Interaction.*

Interaction between objects is handled by the Event Predictor objects and the Events themselves. The *execute_event* method passes pertinent information between the Players involved and calls the appropriate methods in the Players to complete the execution of the Event.

4.4.2.2 *Internal Interaction.*

This model does not specifically address the internal interaction of components of an aggregate object. The model considers such internal interaction to be handled by the Player object itself. Internal Events will be predicted by the Player object and will be executed by the Player when they occur. The exception to this rule is if the simulation modeler chooses to make all the internal components of an object separate objects and have their interaction handled by the system.

4.4.3 *Object Management.*

All objects are maintained in the Player Container of the Partition in which it resides. All objects maintain their own position and velocity information. The map object is used to map relationships between objects. For example, the partitions must be mapped to LPs, and the Players must be mapped to partitions.

4.4.4 *Handling Parallelism.*

Parallelism is handled by using remote Player copies and the relationship mapping system. A Player that is in more than one partition has a copy in each partition that it is in. The position of these multiple partitions may result in a Player having copies on more than one LP.

The Player copy that is in the partition that contains the center of the Player is called the owner copy. All others are simply referred to here as Player copies. Each LP has its own event queue, which is shared between all the partitions on that LP. To eliminate the possibility of duplicate predicted events, each Player copy can only predict Events that will occur in the partition that the

Player copy resides in. The owner Player copy is responsible for predicting all internal Events and all boundary crossing Events. This approach assumes that a Player copy will not be large enough to reach entirely across two partitions in width.

Once an Event is predicted, it is scheduled on the next event queue on the logical processor containing the partition in which it was predicted to take place. This is necessary because the affected Players reside on that processor, or the Event would have been predicted by the owner processor.

The Event is executed in the partition in which it was predicted, and on completion, once the Player position is updated, the Player sends Player updates to the owner copy, which updates itself and sends Player copy updates to all its copies. This sending of Player updates keeps all the Player copies up to date with the same data.

The other problem caused by parallelism is the problem of keeping Events occurring in the correct sequence across all the processors. For example, a Player update Event received from another processor may cause a scheduled Event on the current processor to no longer be appropriate due to a change in the state of the Player object. The solution to this problem is to verify Events before executing them to make sure that they should still happen. This can be done by comparing the states of the objects involved in the interaction and ensuring that conditions for the interaction are met. An example of verifying a collision Event is to make sure that the two objects involved are a minimum distance apart (zero or near-zero) at the time of the collision and are heading toward each other. All update Events from other processors should occur as scheduled, since the Events that caused them have already occurred on other p ers.

4.5 Changing the Simulation

The simulation object interaction is designed so that changing the simulation requires minimum modification to the parts of the model. Modification to the simulation involves adding or modifying Player and Event objects, as well as the Event Predictor objects that support the Events.

4.5.1 Adding Players. Adding a new simulation Player requires adding the new Player subclass with the appropriate methods and modifying the appropriate Relationship Maps. If the Player requires Events that are not already part of the simulation, the simulation must also provide the Event and Event Predictor objects that are required. If these objects already exist, the Relationship Maps must be adjusted to map the new Player type to the correct Event Predictors and map the Event predictors back to the Player.

4.5.2 Adding Events. Adding Events requires adding in the appropriate Event subclass and Event Predictors. Then the appropriate Relationship Maps must be adjusted to map Player classes to Event Predictors and Event Predictor classes to Player Classes.

4.6 Design Decisions

This section discusses the design decisions that influenced the design of the simulation model, as related to the discussion in the previous chapter. The issues of object representation, object interaction, event handling, and parallelism were all an important part of the model's design.

4.6.1 Object Representation. The representation of the objects in the simulation design was influenced by object-oriented design and goals to make the simulation easy to modify. The other goal was to attempt to handle all types of objects the same way in order to avoid complexity of handling multiple types of objects.

4.6.1.1 Simulation Object Representation. In order to keep the simulation easy to modify, all simulation objects are designed so that they can be handled the same way by the main simulation. Therefore, a Player Container object for simulation objects is capable of handling any type of simulation object. To achieve this, a single simulation object class is created that encompasses all simulation objects. All objects in this class must have methods to allow other objects to access and change their states.

4.6.1.2 Aggregate Object Representation. Aggregate objects are represented as objects that are closely coupled or as single objects that control their internal interactions. Components of aggregate objects can be implemented as regular simulation objects. In this case, they can be represented just like any other simulation object, except that there are only certain other objects that they can interact with. The advantage of this design is that it allows for a simpler model by allowing all object communication to be handled the same. Treating internal component objects as a "special" case would require more complicated handling of their communication.

This simulation design also supports aggregate objects that fully control the interaction of their components. These objects must predict their own internally scheduled Events. The advantage of this approach is that all internal component interaction is handled by the object instead of requiring the simulation system to be more complicated. No real design decision was made here since the model supports both methods of implementing aggregate objects and leaves the choice of implementation up to the modeler.

4.6.1.3 Environment Object Representation. Environment objects are represented like any other object, as part of the Player object class. This allows them to interact with other objects using the same method of interaction used by the simulation objects. The problem of how to partition an environment object among simulation partitions is independent of their basic representation. Partitioning decisions determine whether each LP has a whole or partial copy of

an environmental object. Regardless of the choice, the basic representation of the object and its interface methods to the rest of the simulation are identical.

4.6.2 Object Interaction.

4.6.2.1 Object Visibility and Independence . In this model, simulation objects do not have visibility to other objects in the simulation. However, the Event objects and Event Predictor objects do have visibility to all of the simulation objects, since they are the objects that control the simulation. This decision was made to allow for easier updating of the simulation when objects are added and deleted.

4.6.2.2 Simulation Object Interaction. Interaction between simulation objects is taken care of indirectly by the use of Event Predictors and Events. The Event Predictors determine whether two simulation objects will interact. This is done by comparing the states of all objects that qualify for a particular type of interaction to see if an interaction will occur. Actual operation of this process is discussed in Section 4.4. Actual object interaction is handled by the execution of Events. The Events handle all calls to the objects that are involved in the interaction.

4.6.2.3 Aggregate Object Interaction This design supports aggregate object interaction of two different types. Either an object can fully control interaction between its components and schedule these Events on the next event queue, or each of the components can be represented as a Player object and directly interact with Events and Event Predictors.

4.6.2.4 Environment Object Interaction. All interaction between the environment and other simulation objects are handled by Events and Event Predictors. The benefit of this approach is that the Players do not need to know about the environment objects and the environment objects do not need visibility of the Player objects. Their design is therefore independent.

4.6.3 Object Management. The objects in this model manage their own positions in the simulation. The objects themselves are managed by the use of a container object that maintains a pointer to all objects in a given set. The advantage of allowing objects to maintain their own simulation position instead of using a spatial manager is that it minimizes duplication of data in the simulation and minimizes updates that must be made when data is exchanged between LPs.

4.6.4 Event Handling. Event handling in this model is based on the use of discrete-event simulation. The next event for each object must be predicted and scheduled on the event queue. Execution of Events occurs when an Event is removed from the top of the next event queue.

4.6.4.1 Event Prediction. Event prediction is done by a separate entity known as the Event Predictor object. Since simulation objects do not have visibility of other objects, the predictor object must have visibility of all of the objects that it can predict Events for. The Predictor's associations with object classes are mapped by the Relationship Map object. This design allows all interaction of objects to be handled external to the simulation objects, decreasing the visibility requirements of the objects. Additionally, all references to specific Events are encapsulated in the Event Predictors that predict those events. This minimizes the complexity of changes when the application must be modified.

For aggregate objects that have internal components that are not recognized as regular objects, prediction of Events representing interaction internal to the simulation object is performed by the object itself. This minimizes complications to the Event handling system and allows an object to be completely encapsulated.

4.6.4.2 Event Execution. Event execution is performed when an Event is removed from the top of the event queue. The *execute_event* method of the Event object is executed and handles all interaction between the objects. This allows objects to be independent of each other, but requires the Events to have visibility of the objects that it affects.

4.6.5 Parallelism.

4.6.5.1 Logical Processor Synchronization. LPs are synchronized by using the conservative approach of requiring each LP to wait until its input channels are all at a time greater than or equal to the time of the next event queue before allowing another Event to be removed from the next event queue. This design is influenced for this model by the availability of Spectrum, which provides this support.

4.6.5.2 Simulation Object Synchronization. Simulation objects are copied onto all of the LPs that they exist on. Only one of the copies is the main object, and it controls the updates of the copies. This approach was taken to minimize communication between logical processors during Event prediction.

Every object copy performs an Event prediction and schedules its next event on the event queue of the LP that it is on. The Event is also executed at that same LP since the Event may involve updating other objects that are also on that LP. After the Event is completed, each object copy that was modified needs to update its other copies. If the copy is the main copy, it sends updates to all of its copies. If the updated copy was not the main copy, it sends an update to its main copy, which in turn sends updates to each of its copies. The advantage of this design is that only the main object copy is required to maintain visibility of all the copies, while each of the copies must have visibility to their main copy.

4.6.5.3 Simulation Partitioning. This model partitions the simulation spatially. The design minimizes the communication between processors and also limits the search space during Event prediction. This decision was made based on the fact that most of the communication between objects in this type of simulation is interaction with other objects located in the vicinity of the object. Also, this method was already being used in Battlesim and had been used successfully.

4.7 *Conclusion*

This chapter discussed the design of the simulation model and why certain design decisions were made. Chapter V discusses the actual implementation of the model into Battlesim.

V. Implementation

5.1 Introduction

The Battlesim program developed in the AFIT parallel processing laboratory is used to implement this model. This section discusses the selection of the programming language, the requirements for implementation, and the resulting simulation model.

5.1.1 Programming Language. The Battlesim program is written using the C programming language, and that is the programming language selected for this implementation. Analysis during the early stages of this research determined that there was not a C++ compiler available for the Hypercube to use C++ to supplement the current code with object support. Also, the use of Ada would require all of the Battlesim program, including the simulation support provided by TCHSIM and the parallel communication support provided by SPECTRUM to be rewritten in Ada.

The C programming language is not an object-oriented language, but can produce code that supports object-oriented principles. There are several methods to support object-oriented methodologies in C. Matsche presents an approach that implements object attributes within a record structure, class inheritance through the use of `#define` statements to define each record structure that is reused in subclasses, and polymorphism through the use of function pointers to methods within the object record structure (16). The biggest concern here is support of polymorphism, to allow the same method calls to be reused on different types of objects. Matsche recommends using pointers to functions as attributes to each object so that the method can be found directly by using the player pointer. Unfortunately, the method of using function pointers to methods as attributes of each object does not work when objects are transferred to other processes in a parallel simulation. The memory pointers will not be accurate on the new processor.

5.2 *New Battlesim Requirements*

The first new requirement for Battlesim was to overcome the following limitations of the original system:

- The system must be able to handle different types of simulation players and event types, either in the same simulation or to be interchanged to be used in a different simulation with the same simulation support system.
- The interchange of player types and event types must require minimum modification to the simulation support system.
- The simulation must be made more object-oriented.
- The system must be made to execute in parallel.

An additional requirement is to allow the simulation model to support environmental objects, such as terrain and weather, again without causing significant modification to the existing simulation model.

5.3 *Battlesim Implementation*

The simulation model was able to be mapped to Battlesim with a minimal amount of changes to the model. However, extensive modifications to the Battlesim code were necessary in order to implement the new object-oriented structure and the new method of scheduling and handling events. Figures 11 and 12 show the object diagram for the new Battlesim model.

5.4 *Player Implementation*

The player object was implemented differently in the Battlesim implementation because C does not support object-oriented classes easily and the design was altered to simplify the addition of simulation objects to the simulation model. Figure 13 shows the object model for the simulation

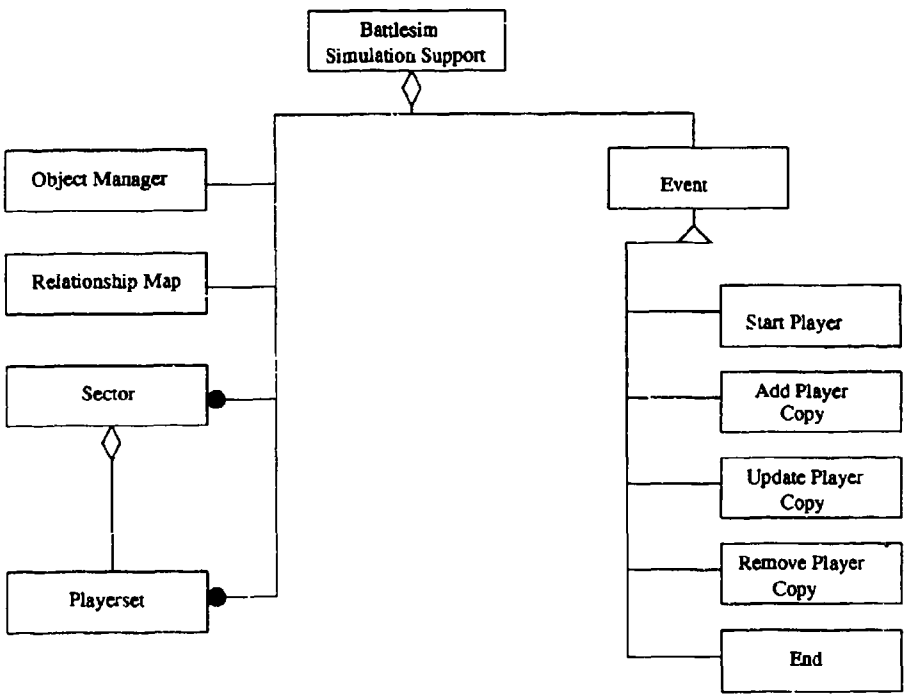

Figure 11. Object Diagram for the Battlesim Simulation Support System

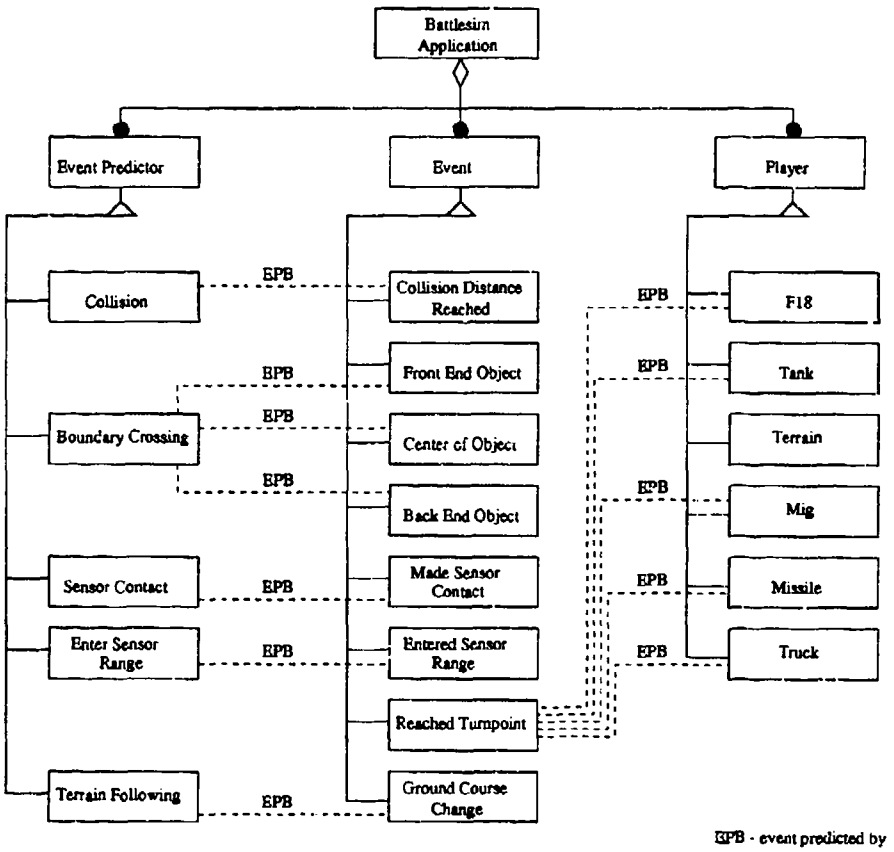

Figure 12. Object Diagram for the Battlesim Simulation Application

implementation of the player. The main Player class contains the object type, id, time, position, velocity, orientation, size, mass, and polygon list that are common to almost any moving object. The subclass for any particular object contains attributes and methods that are specific to the object being simulated. Very simple objects may not need a subclass for their implementation.

Polymorphism is supported by having the player pointer being passed as a parameter in all method calls. For any methods that must be used for more than one player object type, the player object determines which method to use by checking the object type. Each object subclass type must have a differently named method which gets called by the player. Typical methods that are polymorphic are calls for creation, destruction, copying, packing and unpacking objects. File *methods.c* is the Player's polymorphic method call resolver, which checks the object's type and uses a case statement to call the correct subclass method.

5.5 Event Implementation

Events in the Battlesim program are implemented using the event.c file to represent Event object structure and the *ex_event.c* file for the *execute_event* method. The *ex_event.c* file contains a primary *execute_event* method which uses a *switch* (or *case*) statement to select between the event specific *execute_event* methods.

5.6 Event Scheduler Implementation

The Event Scheduler object is implemented in file *schedule.c*. The only function in this file is *det_next_event*, which makes calls to the Player object's internal event prediction method, to the relationship map object, and to the event Predictor Objects. This object has no attributes and is implemented exactly as modelled in the design.

Implementation Player Object Diagram

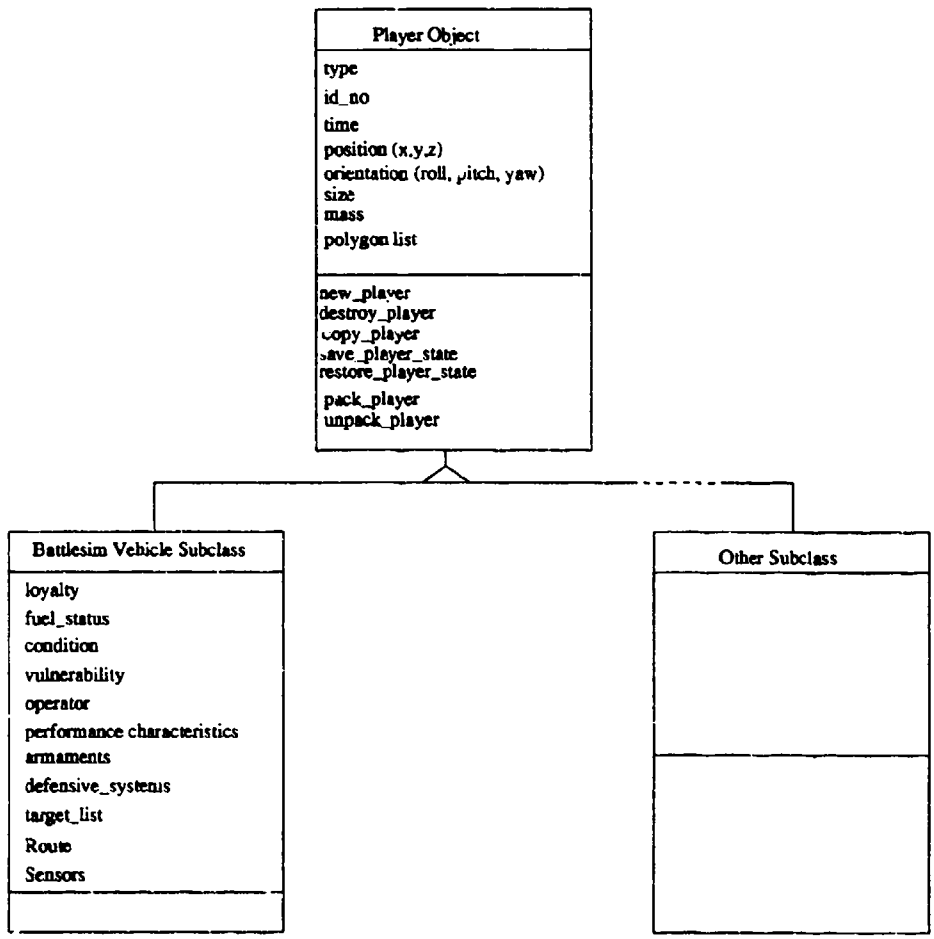

Figure 13. Object Diagram for the Battlesim Simulation

5.7 Event Predictor Implementation

The Event Predictor object is implemented in file *predict.c*. The only externally visible function in this file is *predict_event*. This function's parameters are a pointer to a player and an event predictor type. Based on the predictor type, this function will call the appropriate Predictor Object. This object has no attributes and is implemented exactly as modelled in the design.

5.8 Player Container Implementation

The Player Container object is implemented in file *playerset.c* and is modeled as a Playerset object. The Playerset Object uses a linked list to keep track of the objects that it contains. This file was not changed for this new implementation.

5.9 Partition Implementation

The Partition Object is implemented in file *sector.c* and is represented by a Sector object. Each Sector Object has minimum and maximum X, Y, and Z values that define the area represented by the Sector. Each Sector also has an associated Playerset Object. Minor modifications were made to this file to make it more object oriented.

5.10 Relationship Map Implementation

The Relationship Map object is implemented in file *map.c*. It provides calls that build and modify maps. All of the calls use from none to all of the following parameters:

Map Name The name of the map to be modified.

Map From The item that is being mapped from.

Map To The item that is being mapped to.

The following list shows the maps that are implemented for a basic simulation. Simulation modelers can add other map types that are needed for other applications.

PLAYER_PTR_TO_SECTOR_ID Maps a player object to the sector that it is in.

PLAYER_ID_TO_OWNER_SECTOR_ID Maps a player ID to the sector that its owner is in. This is used for updating player copies.

OWNER_ID_TO_COPIES_SECTOR_ID Maps the owner copy of a player to the sectors where its copies are located. Used for updating player copies.

PLAYER_CLASS_TO_PREDICTOR_CLASS Maps each player class to the predictor classes that predict its events. Used by the Scheduler object to determine which predictors to use when scheduling an event for an object.

PREDICTOR_CLASS_TO_PLAYER_CLASS Maps each predictor class to the player classes that must be used for predicting a specific type of event. Used by the Event Predictors to selectively evaluate player objects from the Playerset.

5.11 Object Copy Manager Implementation

The Object Copy Manager object is implemented in file *obj_mgr.c*. The only externally visible function call is *send_Pcopy_updates*. The only parameter for this function is a pointer to the player that has been updated. The function uses the Relationship map to find the location of the other player copies, creates an *Update_Player* event, and calls *send_event* in file *interfaceB.c* which makes a SPECTRUM call to send the event to another LP.

5.12 Test Cases

The implemented model was tested against several test cases. The first test was to ensure that a single object could travel throughout the battlefield across multiple sectors. The battlefield

was modeled as 8 sectors on 1 LP. This test worked fine with the Player object properly executing all of its boundary crossing events and properly updating its copies as its state changed.

The second test involved testing a single object crossing sectors on a simulation mapped onto multiple LPs. The model used was a mapping of 8 Sectors onto 2 LPs. Once again, the simulation worked well.

The third and final set of tests used 10 objects in a simulation mapping 8 Sectors onto 2 LPs. This test was also a success as it allowed interaction between the simulation objects. Collisions were properly predicted and executed between objects.

5.13 Conclusion

This chapter discussed the implementation of the simulation model into Battlesim. Chapter VI discusses the results and conclusions of the research and recommendations for future research.

VI. Results, Conclusion, and Research Recommendations

6.1 Introduction

This final chapter discusses the results of the research, conclusions reached, and recommendations for further research.

6.2 Results

6.2.1 The Simulation Model . The resulting object-oriented simulation model proved to be effective when implemented into the Battlesim program. The model effectively allowed multiple player types to be simulated within the same simulation system. Also, different object types were allowed to utilize different event types, without interfering with the design of other objects in the simulation. The model also is able to support environment objects such as terrain or weather, although these types of objects have not yet been tested.

6.2.2 Battlesim Program Modification. The implementation of the object-oriented simulation model into Battlesim was a major programming effort. Much of the original simulation model had to be reprogrammed to support object-oriented standards and to support the new event scheduling technique.

One of the lessons experienced from this research came from implementing an object-oriented design using a non-object oriented programming language. There was much extra programming required to implement some of the object-oriented constructs that would not have been required had an object-oriented language been used.

The results were effective in transforming Battlesim into a much more flexible simulation program. Although substantial program changes were required to be made to Battlesim to implement the model, the new simulation model provided a method of adding new object types to the simulation with only minor adjustments to the simulation.

The new event scheduling system allows objects to schedule internal interaction events. This allows aggregate objects to schedule events on the event queue for their component objects. The new event scheduling system also allows player object classes to each be assigned a certain set of event types that apply to objects in their class. This new method of predicting events in the new model made the Battlesim software more efficient by causing the program to only use event predictors that were applicable to the object that was being predicted.

The relationship map provides a mapping for any object to any other object. The map is dynamically modifiable during program execution and allows objects to be implemented without pointers to other objects. The map allows modifications for different mappings so that the simulation modeler can add other maps that need to be represented in the system

6.3 Conclusions

The resulting simulation model is very effective for an object-oriented discrete event simulation. The model allows flexibility for multiple types of objects and object interactions and communication between object copies on different LPs. The model also provides the capability to easily modify the simulation for other applications.

The technique of providing an Event Scheduler, Event Predictors, and allowing events to execute themselves proved to be a promising method of designing a modifiable object-oriented discrete event simulation. The model proved to successfully model the interaction between simple objects. Since the model was designed to allow environmental objects to be modelled the same as simple objects, the model also provides an effective approach to modelling the interaction of simple objects with environment objects. This ability was not tested, but can be shown by implementing terrain represented as a simple object and interacting with other objects through the use of Event Predictors and Event objects just like those used for simple objects.

Aggregate objects can also be represented with the model. The components of an aggregate object can either be implemented as simple objects that are tightly coupled to other components or the aggregate can be modeled as a single object which maintains its component objects and handles the interaction between them.

The simulation model is a valid approach to parallel discrete event simulation. The model maintains communication between LPs and the updates between multiple object copies. One of the problems encountered in the implementation of the model was how to ensure valid events on the event queue for a particular object when events are received from object copies on other LPs. This problem is common to any model of parallel discrete event simulation that predicts the next event for each object individually instead of predicting the next event for any object. This latter method allows only one event to be predicted at a time and does not need a queue.

Although the model proved to be successful, implementing the design in a non-object-oriented language proved to be a very complicated undertaking. I do recommend the design technique, but only if it is to be implemented in an object-oriented language. The object-oriented principles of inheritance and polymorphism cannot be easily reproduced in a non-object oriented programming language without a complicated implementation.

This study in object oriented simulation is a good basis for understanding the problems involved in designing and implementing an object oriented parallel discrete event simulation, and provides a valid simulation model that supports interaction between all types of objects. J-MASS can use this document as a reference for identifying problems in designing a simulation model and use the designed model as an example simulation model design.

6.4 Research Recommendations

There is still much work that can be accomplished in the topic of this thesis. The following lists of items are recommendations for further research in the topic and support of the Battlesim program.

6.4.1 Further Research.

- Investigate the effects of load balancing of a partitioned simulation by modifying the partition structure if the processing load of one of the processors is bearing a significantly larger processing load than the other partitions.
- Investigate further the handling of aggregate objects. Determine if it is better for component objects to be hidden from the simulation and handled by the player or for the simulation to directly handle communication between the component objects.
- Further study the implementation of environment objects. Many types of environment objects can be partitioned for use in a partitioned simulation, but may not be valid if a simulation uses dynamically changing partitions.
- Further study the best method to keep parallel discrete event simulations synchronized when duplicate objects on multiple LPs must keep each other state updated. The problem to be studied involves choosing between taking invalid events off the event queue or verifying each event when it is removed from the queue by ensuring that the current state of the objects allows the event to occur.

6.4.2 Battlesim Updates.

- Implement Battlesim using an object-oriented language. This would allow better use of object classes and polymorphism.

- Modify the *sector* to *LP* mapping which uses *tchmap.c* to use the new map system in *map.c* to provide more consistency to the code.
- Change the object update event to send partial updates instead of entire player copies. This would lower the communication between processors during object updates.
- Make the Battlesim subclass more object-oriented by taking the armaments, defenses, target lists, and sensors out and making them into separate objects like the route.
- Add an acceleration model into the player objects. The objects currently change velocity instantly.
- Modify the use of scenario files to be able to load a single scenario file instead of one for each section. This would require a function to place an object into the correct sector(s) based on its location and to be able to send the object to the other LPs if necessary. This would simplify the process of generating scenarios.
- Modify Spectrum to use float times for LP channel updates instead of integer times. Battlesim events occur at float times and synchronization between partitions may not be accurate when objects are migrating to other processors.

Appendix A. Model Data Dictionary

A.1 General Model

A.1.1 General Model Objects.

Application - The Application is the part of the simulation model that contains the objects being simulated, the event predictors that predict object interactions, and the events that support the interactions.

Application Support System - The Application Support System provides support for a simulation application. The support objects include the Event Scheduler, the Object Copy Manager, the Object Relationship Map, and simulation support events.

Event Object - An Event object signifies the occurrence of an object interaction or the change of state of an object. Events can be of two different types: application events and simulation support events. Application events signify object interaction or change of an object's state and are created by an event predictor. Simulation support events are created by the simulation in order to carry out a particular action at a particular point in time. Events maintain pointers to the objects that they affect and have an event type that determines what actions are taken when their *execute_event* function is called.

Event Predictor Object - An Event Predictor computes the next occurrence of a specific event type for a given Player object.

Event Scheduler Object - The Event Scheduler determines the next event for a given Player object by consulting all of the Event Predictors that correspond to that Player object.

Object Copy Manager Object - An Object Copy Manager manages the update of object copies by determining the location of the copies and sending *Update_Player_Copy* events to the appropriate LP's.

Object Relationship Map Object - An Object Relationship Map maintains the mapping of relationships between objects since the simulation objects in this model do not have visibility of other simulation objects.

Partition Object - An Partition Object is a 3-dimensional partition of the simulation environment.

Player Container Object - A Player Container object is a container object that consists of a list of objects. For this model, a Player Container Object is associated with each Partition Object and contains a list of all of the objects located in that simulation partition.

Simulation LP - A Simulation LP is a logical process of the simulation. Each Simulation LP consists of both Simulation Support and Application Support.

Simulation Support System - The Simulation Support System in this model supports an application using Parallel Discrete Event Simulation. The support system implements discrete event simulation support by providing a simulation clock and a next event queue. Parallelism is supported by providing an interface that makes parallelism invisible to the application.

A.2 Battlesim Model

- **Event Predictor Objects**

  Boundary Crossing - The Boundary Crossing Event Predictor predicts events for Player Objects crossing the edges of sectors. This event predictor only applies to moving Player objects in the simulation. The Events predicted are Front End Object, Center of Object, and Back End Object.

  Collision - The Collision Event Predictor predicts collisions for certain Player Objects with other Player objects. The predictor is only used by Mobile Player Objects that have the capability to collide with other objects. The event predicted is Collision Distance Reached.

Enter Sensor Range - The Enter Sensor Range Event Predictor predicts when a mobile Player Object moves into the sensor range of another player object that has a sensor. The event predicted is Entered Sensor Range.

Sensor Contact - The Sensor Contact Event Predictor predicts when a Player Object that has a sensor will have another player object move into its sensor range. The event predicted is Made Sensor Contact.

- **Event Objects**

Back End Object - The Back End Object event indicates that the back end of an object has left a sector. The player will be removed from that sector and all appropriate object relationship maps will be modified to eliminate a reference to that sector as the location of a player copy.

Center of Object - The Center of Object event indicates that the center of an object has moved to a new sector. The Relationship Maps must be updated to reflect that the owner player has moved to a new sector and the mapping to player copies must be made available on the LP where the owner player is now located.

Collision Distance Reached - This event indicates that a two player objects have collided. The event exchanges information between the players to indicate the speed and mass of the object they have collided with and the players respond accordingly.

Entered Sensor Range - This event indicates that a player object has entered the sensor range or another player. The event notifies the player through a method and the player responds accordingly.

Front End Object - The Front End Object event indicates that the front end of an object has just entered a new sector. A copy of the player must be created in the new sector and all appropriate object relationship maps will be modified to indicate the player now has a player copy in that sector.

Made Sensor Contact - The Made Sensor Contact Event indicates that a player's sensors have made a contact. The event will indicate the contact to the player by calling a method in the player which will respond accordingly.

Reached Turnpoint - The Reached Turnpoint Event indicates that a player has reached one of its turnpoints. The event will call a method in the player which will handle this event.

Remove Player Copy - The Remove Player Event indicates that a player copy is no longer in a given simulation sector and that it must be removed. This may occur as the result of a player leaving a sector or a player being destroyed.

Start Player Copy - The Start Player Event indicates that a player needs to have its next event calculated. This Event is mainly used only at he beginning of the simulation to calculate the first event for each of the player objects.

Update Player Copy -This event indicates that the player copy must be updated using the player copy attached to the event. If the player does not exist in this sector, then it must be created.

- **Player Objects**

F18 - This player is a mobile player that flies throughout the simulation environment, from routepoint to routepoint, reacting to other objects that it encounters.

MIG - This player is a mobile player that flies throughout the simulation environment, from routepoint to routepoint, reacting to other objects that it encounters.

MISSILE - This player is a mobile player that flies throughout the simulation environment, from routepoint to routepoint, reacting to other objects that it encounters.

TANK - This is a mobile player that moves along the terrain in the simulation, from routepoint to routepoint, reacting to the terrain and other objects it encounter.

TERRAIN - This object is a stationary environmental object that exists throughout the simulation. In this model, the terrain does not change.

- Other Simulation Objects

  Map - The Object Relationship Map for Battlesim. Keeps track of associations between objects.

  Object Manager - The Object Copy Manager for Battlesim. Manages updates of player copies by determining what LP object copies are on and sending *Update_Player_Copy* events to those LP's.

  Playerset - The playerset is a Player Container Object and contains all the Player Objects in a given Sector.

  Sector - The sector is a Partition Object and represents a partition of the Battlesim simulation environment.

Appendix B. Detailed Player Object Attribute Descriptions

This appendix describes the updated Player Object in BATTLESIM by defining the attributes of the Player object their purpose. The first section describes the attributes of the Player Class, which is the generic player structure used for all simulation players in this model. The second section describes the attributes of the the Battlesim Player Subclass.

Note that if an attribute is not actively used in BATTLESIM, then it is marked as such; these attributes remain to support future improvements in simulation fidelity.

B.1 Player Class Attributes

This section describes the attributes of the Player Class.

- **Object Type** - identifies what kind of icon is associated with a player when the scenario's output file is displayed by the graphics display driver.
- **Object Identifier** - unique among "owned" players; however, player-copies of an owned player will share the same object identifier.
- **Current Time** - current *simulation time* of the player
- **Location** - three values indicating the position of the player on the battlefield in x,y,and z coordinates.
- **Velocity** - three values indicating the player's x,y, and z velocity vectors.
- **Orientation** - yaw, pitch and roll of player (roll not used in current version).
- **Rotation** - rates of changes about the x, y, and z axis'. These values are not used by the current version.
- **Player Size** - attribute indicating the radius of the player in the same units that are used to specify position. This is used to make collision detection more realistic.

- **Player Mass** - attribute indicating the mass of the player. This is used to make collision response more realistic.

B.2 Battlesim Player Subclass Attributes

This section describes the attributes of the Battlesim Player Subclass.

- **Object Loyalty** - a number indicating which objects are friends and which are foes. Objects with the same number are friends, and will not attack each other. Objects with different numbers are foes, and may attack one another.
- **Fuel Status** - value indicates how much fuel the player has left. Not used in the current version.
- **Condition** - value indicating how badly damaged a player is. Not used in the current version, since an object is alway either fully operational or completely destroyed.
- **Vulnerability** - specifies how strong a destructive force is required to destroy the player. Not used in the current version.
- **Operator** - two values indicating the experience and threat knowledge of the operator of the player object. Not used in the current version.
- **Performance** - four characteristics of the player not used in the current version. These characteristics include:

  1. max speed
  2. minimum turn radius
  3. average fuel consumption rate
  4. max climb rate

- **Route Data** - a pointer to the route list.

- **Sensors** - a pointer to the linked list of the sensors owned by a player. The characteristics of a sensor include:

  1. sensor type (not used by current version)
  2. sensor range
  3. sensor resolution (not used by current version)

- **Armaments** - a pointer to the linked list of the armaments owned by a player. The characteristics of an armament include:

  1. type (used by missiles only)
  2. range (not used by current version)
  3. lethality (not used by current version)
  4. accuracy (not used by current version)
  5. speed (not used by current version)
  6. count (not used by current version)

- **Defensive Systems** - a pointer to the linked list of the defensive systems owned by a player. It is not used by the current version. The characteristics of a defensive system include:

  1. type
  2. range
  3. effectiveness

- **Target List** - a pointer to the linked list of targets for the player. The characteristics of a target list include:

  1. type
  2. location (not used by current version)

Appendix C. Simulation Model User Guide

This section provides instructions on how to add new Players and Events to the simulation model.

C.1 Adding New Player Types

C.1.1 Modified Files To add new Player types to the simulation, modifications must be made to the following files.

player.h This file must be changed to assign the new Player type a numeric value.

subclass.c and subclass.h The subclass files must be added to provide the structure and methods for the new Player subclass. For example, the subclass files for Battlesim are *bs_player.c* and *bs_player.h*.

methods.c The methods in this file must be updated to be able to handle calls to the new Player subclass.

init_maps.c This file must be changed to associate the new Player type to the Event Predictor classes that it is associated with and associate Event Predictor classes to the new Player type.

scenario files The modeler must create a new format for the new Player subclass and generate scenario files. (Only necessary if the subclass has its own data structure.)

C.1.2 Required Methods. New Players are required to have variations of the following methods defined:

new_subclass Allocates space for the subclass structure and returns a pointer to the new subclass. (Only necessary if the subclass has its own data structure.)

free_subclass Frees space occupied by a subclass structure. (Only necessary if the subclass has its own data structure.)

read_subclass Reads in data from the scenario file to the subclass structure. (Only necessary if the subclass has its own data structure.)

list_subclass Lists the data in the subclass data structure. (Only necessary if the subclass has its own data structure.)

pack_subclass Packs the data of the subclass data structure into newly allocated contiguous memory and returns a pointer to the memory. (Only necessary if the subclass has its own data structure.)

unpack_subclass Unpacks data from contiguous memory into a subclass data structure and returns a pointer to the structure. (Only necessary if the subclass has its own data structure.)

subclass_packsize Determines the memory required to pack a subclass data structure. (Only necessary if the subclass has its own data structure.)

det_subclass_internal_event Determines the next internal event for the particular Player subclass. For example, the Battlesim Player uses this to predict route events. (Only necessary if the subclass schedules events.)

C.2 Adding New Events

Adding new Events to the simulation requires modifications to the following files:

ex_event.c The proper "execute_event" method must be added to this file and the polymorphic *execute_event* method must be updated to handle it.

ex_event.h The new Event type must be assigned a numeric designator.

predict.c The proper "predict_event" must be added to this file and the method *predict_event* must be updated.

predict.h The new Event Predictor type must be assigned a numeric designator.

init_maps.c This file must be changed to associate each Player class to the Event Predictor classes that it is associated with and associate each Event Predictor to the Player classes that are in its search space.

C.3 Example Implementation of a Pool Ball Simulation

This example will describe how the simulation can be made to simulate a set of pool balls bouncing around a pool table. The balls are able to bounce off of each other, as well as the sides of the pool table. The pool table will be partitioned just as the battlefield was. Each Sector object will therefore represent a different portion of the pool table. Figure 14 shows an example Pool Ball simulation.

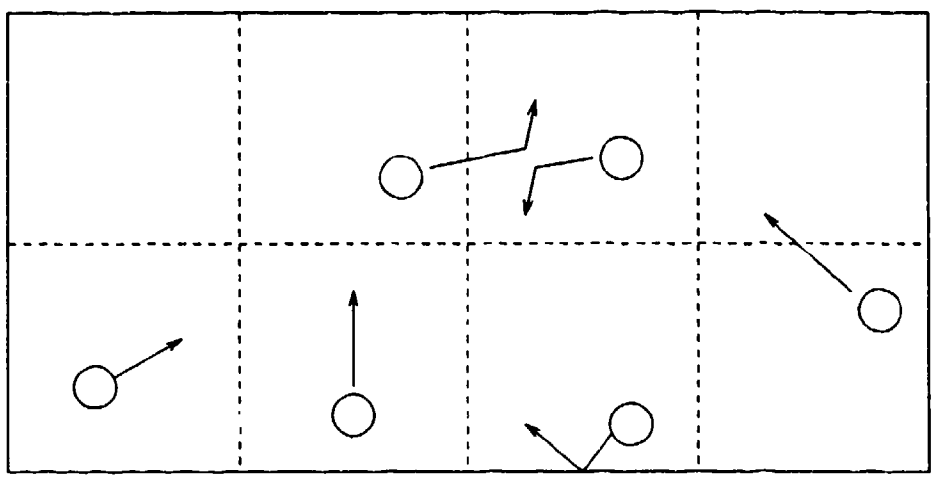

Figure 14. Example Pool Ball simulation.

C.3.1 Description of the New Objects. The objects involved in the pool ball simulation are pool balls and the pool table. The only Events required for this simulation, besides the simulation maintenance Events, are Boundary Crossing Events and Collision Events

C.3.1.1 The Pool Balls. The pool balls will be modeled as spheres with a certain size and mass. The attributes for the pool balls can be implemented using the standard simulation class provided in file *player.c*. Each ball can be represented using the size and mass fields in the Player structure which are entered as part of the scenario file. No polygons are necessary for collision detection since these are regular shaped objects.

C.3.1.2 The Pool Table. In the simplest model, the pool table can be implemented as a set of planes. Since the balls do not react to gravity, the bottom plane of the table does not need to be modeled. What does need to be represented is the sides of the pool table, because the pool balls must interact with them as a Collision Event and bounce off of them. The easiest way to do this is to represent each of the sides of the pool table using a plane equation, providing A, B, C, and D for the equation $Ax + By + Cz + D = 0$. Since the sides of the pool table are square with the Cartesian coordinate system, this will result in the four sides being represented as (1,0,0,-L), (1,0,0,-R), (0,1,0,-F), and (0,1.0,-B), where L, R, T, and B stand for left, right, front, and back. The representation of each side must exist in the scenario file once for each Sector that the side must reside in. Each of the sides must have an X, Y, Z location in the main player class that is set to a value within the range of the Sector it resides in. Otherwise, it will not be represented in the correct sector.

C.3.2 Adding the New Objects. This section will step through the actions required to implement the new objects. Each file to be modified is listed, as well as the changes that must be made.

player.h This file contains the assignments of numeric values to Player types. It must be modified so there can be a numeric representation of a Pool Ball object and a Pool Table Side object.

Example addition to the file:

```
#define POOLBALL 6
#define POOL_TABLE_SIDE 7
```

subclass.c and subclass.h The Pool Ball object requires a subclass file that contains a method that handles its reaction to a collision, but does not require a subclass data structure since all of the necessary attributes are available in the general Player object class. The collision handling method must be able to take in the parameters that describe the impact and change the state of the Pool Ball as a result of the impact.

The Pool Table Side object requires a subclass file that contains its attribute data structure. This subclass does not need interaction methods. Since it has a data structure it must have the required methods to handle the data structure. These include versions of the following methods:

- new_subclass()
- free_subclass()
- read_subclass()
- list_subclass()
- pack_subclass()
- unpack_subclass()
- subclass_packsize()
- pack_subclass()

These methods must be given names that are unique in the simulation. An example would be *new_PTableSide_subclass*. Even though the Pool Table Side object will not be moving between partitions, the pack and unpack functions are still required. This requirement is needed to support the simulation's capability to save its state and return to an earlier simulation time. All object structures must be able to be packed.

methods.c The methods in this file must be updated to be able to handle calls to polymorphic methods in the new Pool Ball and Pool Table Side subclasses. The collision handling method

in the Pool Ball subclass must be added in the *execute_collision* method. The Pool Table Side does not need to have an entry in the *Execute_Collision* method as long as the default entry performs no action. Additionally, all of the Pool Table Side object's subclass data structure handing methods must be added.

init_maps.c If only one Event Predictor object is used to model both Ball-Ball collisions and Ball-Side collisions, then this file must be changed to associate the Pool Ball Player type with the Boundary Event and Collision Event Predictors. Also, the Collision Event Predictor must be associated with both the Pool Ball Class and the Pool Table Side Class. This is so the Predictor will evaluate the state of objects of both of these Player classes when checking for collisions. It may be desirable to implement the Event Predictors for the two types of collisions separately. In this case, there would be a separate Event Predictor of Ball-Ball and Ball-Side collisions.

scenario files Since the Pool Table Side object requires an additional data structure for its subclass, the modeler must design an input format for the subclass and a method for reading it in. The best way to do this is to use a one-line format immediately after the line used for the main Player object.

C.3.3 Addition of the New Events. Assuming the Boundary Events and the Collision Events already exist, no new events are required. The Event Predictor for the Collision Events must be modified to handle Ball-Side collisions. It may be easier to develop a separate Event Predictor for each type of predictor. The following modifications must be made for implementation of the Events and Event Predictors of the Pool Ball Simulation:

ex_event.c and ex_event.h Since no new Events are introduced, these two files do not require changes.

predict.c The *det_collision_event* must be modified to handle two different types of collision. The first method involves predicting the collision of two spheres while the second involves predicting the collision between a sphere and a plane.

predict.h Since no new Event Predictors are needed, this file does not require changes.

init_maps.c This file must be changed to associate the Collision Event Predictor class with the Pool Ball class and the Pool Table Side class.

Appendix D. The Map Object

D.1 Introduction

This appendix describes the Map Object that was created during this thesis. The Map Object contains mappings of an integer to a set of integers. The Map Object uses the linked list support in *ll.c* to create linked lists. File *map.h* is used to define the maps that are used.

D.2 Map Structure

The data structure for the Relationship Map object consists of an array of pointers to maps. Each map consists of a linked list of items that are mapped from. Each of the items that are mapped from consist of linked list of items that they are mapped to. All linked lists are implemented using the linked list facility in file ll.c. Each linked list consists of headers that point to list nodes. Each list node is a list placeholder that has a pointer to a data element and to the next list element if one exists. See Figure 15 for a pictorial representation of the map system.

The following code displays the structures used in the Relationship Map object:

```
static void *map[MAX_MAP_TYPE + 1];

typedef struct {
  int mapfrom_id;
  void *mapping_ll;
  void *last_read_mapto;
} mapfrom_type;

typedef int mapto_type;
```

D.3 Methods

D.3.1 Public Methods. The following methods are public and available external to the *map.c* file. They are made available through the use of file *map.h*.

new_Maps Initializes all the Maps by creating the list headers for each one.

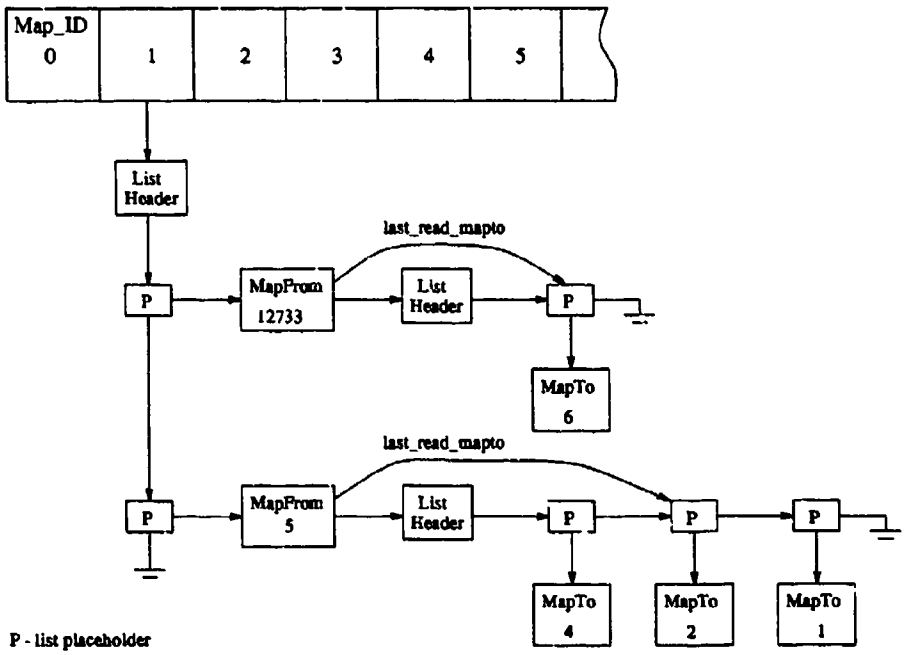

Figure 15. The data structure representation of the Relationship Map

set_Mapping Adds a mapping for a particular integer in a particular map. NOTE: This function allows an integer to be mapped to more than one other integer.

reset_Mapping Replaces all of the current mappings for an integer in a map with a single new mapping.

get_first_Mapping Gets the first mapping for an integer in a map. Returns an integer.

get_next_Mapping Gets the next mapping for an integer in a map, assuming no interim accesses were made to the map. Returns an integer.

is_Mapped Determines whether a mapping exists in a particular map. Returns TRUE or FALSE.

del_Mapping Removes a particular mapping from a map.

del_object_Mappings Removes all of the mappings for a particular integer in a map.

show_all_Maps Shows the contents of all maps.

pack_Mapping Packs a single mapping and returns a pointer to the packed mapping.

get_packed_Mapping_size Gets the size of a packed mapping.

unpack_Mapping Unpacks a mapping and puts it in the map on the current LP. The packed mapping is freed.

show_packed_Mapping Show the contents of a packed mapping

pack_Mapset Pack the entire set of maps. Returns a pointer to the packed maps.

record_Mapset Records the current Mapset. Used by state saving process to pack the set of maps.

restore_Mapset Restores a recorded mapset.

clear_Mapset Clears the entire set of maps.

free_packed_Mapset Frees a packed set of maps.

show_packed_Mapset Show the contents of a packed set of maps.

D.3.2 Private Methods. The following methods are not available externally to the *map.c* file. They are not made available by file *map.h*.

new_mapfrom Creates a new *mapfrom* structure and initializes its attributes. The *mapfrom* structure consists of an integer that is being mapped from and a pointer to a linked list of items that it is mapped to. It also contains a pointer to the last mapto item in the list that was accessed. This pointer is necessary to be able to iterate through the list using the method *get_next_mapping*.

new_mapto Creates a new *mapto* structure and initializes its attributes. The *mapto* structure consists of an integer that is being mapped to.

free_mapfrom Frees a *mapfrom* data structure.

free_mapto Frees a *mapto* data structure.

mapfrom_is_equal Used for calls to the *ll_delete*, *ll_contains*, and *ll_index_data* methods of file *ll.c* to compare a *mapfrom* data node against an entered integer.

mapto_is_equal Used for calls to the *ll_delete*, *ll_contains*, and *ll_index_data* methods of file *ll.c* to compare a *mapto* data node against an entered integer.

D.4 Map Setup and Operation

D.4.1 Map Setup. The first requirement to setting up the Relationship Map is to setup the defined Map Types in file *map.h*.

```
/************************ MAP TYPES ******************************
* IMPORTANT: MAX_MAP_TYPE must be >= largest defined MAP_TYPE    *
*  and all MAP_TYPES MUST BE different.                          *
*****************************************************************/

#define PLAYER_TYPE_TO_PREDICTOR_TYPE  1
#define PREDICTOR_TYPE_TO_PLAYER_TYPE  2
#define PLAYER_PTR_TO_SECTOR_ID  3
#define PLAYER_ID_TO_OWNERS_SECTOR_ID 4
#define PLAYER_ID_TO_COPIES_SECTOR_ID 5

#define MAX_MAP_TYPE  5

/*****************************************************************/
```

The five map type listed above are required for the basic operation of the simulation system. New map types can be added, but notice that the variable **MAX_MAP_TYPE** must be adjusted accordingly. IMPORTANT: The map system is implemented so that space will be allocated and maps initialized for map IDs in the range 0 to **MAX_MAP_TYPE**, so plan your numbering system accordingly.

D.4.2 Map Initialization. All of the maps are allocated space and initialized through the method *init_Maps* in file *init_maps.c*. This method makes a call to *new_Maps* to set up the maps and then uses the other Map methods to load the maps with initial values.

D.4.3 The Three Player Object Position Maps . The three Maps used for maintaining the Player objects in the simulation are listed below:

- PLAYER_PTR_TO_SECTOR_ID
- PLAYER_ID_TO_OWNERS_SECTOR_ID
- PLAYER_ID_TO_COPIES_SECTOR_ID

The PLAYER_PTR_TO_SECTOR_ID map is used to maintain the relationship between each copy of a Player object and the ID of the Sector object that it resides in. This map must be maintained for each copy of a player, using the Player's pointer as the map-from value and the Sector ID as the map to value in the Map method calls.

The PLAYER_ID_TO_OWNERS_SECTOR_ID map is used to maintain the relationship between a Player ID and the ID of the Sector that the owner copy of the Player resides in. Only one copy of this mapping is maintained on each LP that a Player Object has copies on, therefore this mapping is not deleted when a Player leaves a sector, in case the Player has copies in other Sectors that are on the same LP. A more complex implementation would determine if it is safe to delete the mapping by checking the other sectors on the same LP to see if they contain a Player object with the same ID. This map can be used to determine if a copy of a Player object is the owner copy by comparing this mapping against the PLAYER_PTR_TO_SECTOR_ID map to see if they are equal.

The PLAYER_ID_TO_COPIES_SECTOR_ID map is used to maintain the relationship between the owner copy of a Player object and its other copies. This is done by mapping the Player ID of the Player to the Sector ID of the Sectors that contain copies of the Player. This map is

maintained only on the LP that contains the owner copy of the Player. This map must be transferred between LPs with the ownership of the Player. This map is primarily used to update Player copies when the owner copy changes.

Appendix E. Scenario Input File Format

E.1 Introduction.

This section describes the format and attributes for a scenario input file.

To allow for the simulation of multiple types of objects, the structure of the scenario file needed to be changed. The old scenario file provided the capability to read in a single type of object. The goal in the design of the new scenario file was to support most of the old design to enable the reuse of code, and to allow for the addition of other object types.

The new design centers around the use of a single data line used for all types of objects, followed by further data lines that are dependent on the type of object that is being entered. The first data line contains data that is common to all types of simulation objects and fills the attributes of the simulation object superclass. The following lines contain data that are relevant to the subclass defining the object. The function of reading in a player from the scenario file has been moved to the player, which reads in the first line, and then calls the method that reads in the subclass data.

E.2 Scenario Input File Format.

The format for scenario files with the '.in5' extension is shown below. Attributes which existed in the previous version of BATTLESIM and have not changed format are merely listed — additional information on their purpose can be obtained from Soderholm's thesis(30:3-5). New attributes added to support spatial partitioning, as well as old attributes with a new format, are explained in detail.

E.2.1 The Header Section.

- **Version Number** - A *character* attribute indicating what version this file is. If the file version does not match that expected by BATTLESIM, then an error message is returned to the user and the run terminates.

- **Terrain Data Filename** - A *character* attribute that indicates the name of the terrain data file to be used for this simulation run. This file contains terrain elevation data which currently is unused by BATTLESIM. However, this was added to support modeling of terrain at a later time.

- **Terrain Min Coordinates (x,y,z)** - Three *double* attributes which provide the minimum x, y, and z axis coordinates of the battlefield. Standardized terrain data files use minimum values of zero.

- **Terrain Max Coordinates (x,y,z)** - Three *double* attributes which provide the maximum x, y, and z axis coordinates of the battlefield. Standardized terrain data files use maximum values of 117,000 for the x-axis and 118,000 for the y-axis. A maximum of 1000 is used for benchmark scenario files in the z-axis.

- **Number of Sectors** - An *integer* value indicating the number of sectors to be used in the scenario. The value must be greater than 0 and less than 65 to be valid.

- **Sector min/max Coordinates** - *Double* values which contain each sector's minimum x/y/z coordinates and maximum x/y/z coordinates, in that order. Each line contains six entries containing the boundary information for a particular sector, with the lines appearing in order from the first to the last sector. Therefore there are exactly as many lines in this section as there are number of sectors.

- **Number of Icon Definitions** - An *integer* value specifying how many icon definitions exist. This value is always five now, since five different types objects can be created in the scenario. This information, while it previously existed, was hard-coded into BATTLESIM.

- **Icon Definitions** - Two attributes, an *integer* and a *character*, which together uniquely describe an icon definition needed to support creation of the display driver datafile using the format previously defined by DeRouchey (7). The five definitions used by current version of BATTLESIM include:
  - Type 1 - f18
  - Type 2 - mig1
  - Type 3 - missile
  - Type 4 - tank
  - Type 5 - truck

E.2.2 The Player Class Section. This information, while it previously existed, was hard-coded into BATTLESIM.

- **Player Class Attributes** - Fourteen attributes, all residing on the same line in the scenario file, which provide information about a particular player in the scenario, no matter what its Object Type. The attributes, in order, are:
  1. Object Type
  2. Object Identifier
  3. Current Time
  4. Location (x-component)
  5. Location (y-component)
  6. Location (z-component)
  7. Velocity (x-component)
  8. Velocity (y-component)

9. Velocity (z-component)

10. Orientation (yaw rate)

11. Orientation (pitch rate)

12. Orientation (roll rate)

13. Object Size

14. Object Mass

E.2.3 The Player Subclass Section. The Player Class Section is defined differently for each type of object being simulated. The simulation modeler is responsible for designing the Player Subclass Section of the scenario file and providing the code to properly read the data and then return control to the player class to be able to read in the next line of data. The Player Subclass Section begins on the line following the Player Class Section and goes as long as is necessary to read in a subclass.

The format for the Battlesim Player Subclass Section is provided below:

- **Battlesim Player Subclass Attributes** - Ten attributes, all residing on the same line in the scenario file, which provide information about a particular Battlesim player's subclass data in the scenario, to supplement the data in the Player Class Section. The attributes, in order, are:

  1. Object Loyalty

  2. Fuel Status

  3. Condition

  4. Vulnerability

  5. Operator (experience)

  6. Operator (threat knowledge)

7. Performance Characteristics (minimum turn radius)

8. Performance Characteristics (max speed)

9. Performance Characteristics (average fuel consumption rate)

10. Performance Characteristics (max climb rate)

- **Number of Route Points** - An *integer* value specifying how many route points there are in the player's **route_data** linked list.

- **Route Points** - Three *double* values indicating the x,y, and z coordinate of one of the player's route points. The points are listed in the order the player goes to each of them. Each line in the file contains exactly one route point, so there are as many lines as there are route points. NOTE: All players that use routes are given their first route point as a starting location, regardless of the information used in the Location attributes in the Player Class section..

- **Number of Sensors** - An *integer* value specifying how many sensors there are in the player's **sensors** linked list.

- **Sensors** - Three *integer* attributes indicating the type, range, and resolution of one of the player's sensors, respectively. Each line in the scenario file contains one sensor, so there are as many lines as there are sensors.

- **Number of Armaments** - An *integer* value specifying how many armaments there are in the player's **armaments** linked list.

- **Armaments** - Six *integer* attributes indicating the type, range, yield, accuracy, speed, and count of one of the player's armaments, in that order. Each line in the scenario file contains one armament, so there are as many lines as there are armaments.

- **Number of Targets** - An *integer* value specifying how many targets (by type or location) are in the player's **targets** linked list.

- **Targets** - One *integer* indicating the type, and three *double* values indicating the x, y, and z coordinate of one of the player's targets, in that order. Each line in the scenario file contains one target, so there are as many lines as there are targets.
- **Number of Defensive Systems** - An *integer* value specifying how many defensive systems are in the player's **defensive_systems** linked list.
- **Defensive Systems** - Three *integer* values indicating the type, range, and effectiveness of one of the player's defensive systems, respectively. Each line in the scenario file contains one defensive system, so there are as many lines as there are defensive systems.

NOTE: Since every Battlesim Subclass player in a scenario must have at least one route point it is starting at as well as trying to reach, no player will have an empty route list at initialization. However, any of the other four player linked lists may be empty. In that case, the linked list in question would have no lines in the scenario file to describe it other than the number attribute.

E.3 Example Battlesim Scenario File

E.4 Benchmark Scenario 13.

This scenario was designed to run with 8 LPs, so 8 scenario files are required to support it. However, since the scenario files for LPs 1 through 7 are identical, only one of them is shown. The required MAP file is next, followed by a diagram depicting the movements of all the players during the entire course of the scenario. This scenario was designed to ensure that a player could correctly cross sector boundaries in the +x, -x, +y, and -y directions in the same scenario ***using all three boundary-crossing events.*** The plane flies a zig-zag pattern to accomplish this.

E.4.1 Scenario Files.

```
**************************************************************************
* FILE: bench130.in5
* AUTHOR: Capt Seth Guanu
* DATE: 17 Nov 93
* DESCRIPTION: This file contains 1 plane description.  Designed to make
*        sure that aircraft are properly replicated when passing from
```

```
*             one sector to another on DIFFERENT LP's.  This file is
*             intended for LP 0.  In this particular scenario one aircraft is
*             started on sector 1 and flies through all sectors.  Each LP
*             has exactly one sector assigned to it (see battlesim13.ma).
*       This benchmark was specifically designed to check for proper
*       scenario execution when players cross sector boundaries in
*       the +x, -x, +y, and -y directions. The plane flies a zig-zag
*       pattern in the x-axis first through all sectors, and then it
*       flies a zig-zag pattern in the y-axis direction through all
*       sectors.
*****************************************************************************
* version number
V5.0
* terrain data filename
terrain.10
* terrain min coordinates (x, y, z)
0.1 0.1 0.1
* terrain max coordinates (x, y, z)
117000.0 118000.0 1000.0
* number of sectors (must be < 64)
8
* sector min/max boundaries (x,y,z values in order from 1st to last sectors)
0.1 59000. 0.1 29250. 118000. 1000.
29250. 59000. 0.1 58500. 118000. 1000.
58500. 59000. 0.1 87750. 118000. 1000.
87750. 59000. 0.1 117000. 118000. 1000.
0.1 0.1 0.1 29250. 59000. 1000.
29250. 0.1 0.1 58500. 59000. 1000.
58500. 0.1 0.1 87750. 59000. 1000.
87750. 0.1 0.1 117000. 59000. 1000.
* number of icon records
5
1 f18
2 mig1
3 missile
4 tank
5 truck
* Player Object
1 1 0 0 0 0 1000 0 0 0 0 0 1 2000
1 1 1 1 1 1 1 1 1 1
* number of route points
13
* route coordinates x,y,z (start to finish order)
8775. 110133.33 500.
108225. 110133.33 500.
108225. 7866.67 500.
8775. 7866.67 500.
8775. 102266.67 500.
20475. 102266.67 500.
20475. 19666.67 500.
43875. 19666.67 500.
43875. 102266.67 500.
73125. 102266.67 500.
73125. 19666.67 500.
96525. 19666.67 500.
96525. 102266.67 500.
* number of sensors
1
1 5850 1
* number of armaments
0
* armament descriptions (if above > 0)
* number of targets
3
* target descriptions (if above > 0)
```

```
1 0 0 0
4 0 0 0
5 0 0 0
* number of defensive systems
0
* defensive system descriptions (if above > 0)
* END OF OBJECT
```

```
*****************************************************************************
* FILE: bench131.in5
* AUTHOR: Capt Seth Guanu
* DATE: 17 Nov 93
* DESCRIPTION: This file contains 0 plane descriptions.  Designed to make
*       sure that aircraft are properly replicated when passing from
*             one sector to another on DIFFERENT LP's.  This file is
*             intended for LP 1.  In this particular scenario one aircraft is
*             started on sector 1 and flies through all sectors.  Each LP
*             has exactly one sector assigned to it (see battlesim13.ma).
*       This benchmark was specifically designed to check for proper
*       scenario execution when players cross sector boundaries in
*       the +x, -x, +y, and -y directions. The plane flies a zig-zag
*       pattern in the x-axis first through all sectors, and then it
*       flies a zig-zag pattern in the y-axis direction through all
*       sectors.
*****************************************************************************
* version number
V5.0
* terrain data filename
terrain.10
* terrain min coordinates (x, y, z)
0.1 0.1 0.1
* terrain max coordinates (x, y, z)
117000.0 118000.0 1000.0
* number of sectors (must be < 64)
8
* sector min/max boundaries (x,y,z values in order from 1st to last sectors)
0.1 59000. 0.1 29250. 118000. 1000.
29250. 59000. 0.1 58500. 118000. 1000.
58500. 59000. 0.1 87750. 118000. 1000.
87750. 59000. 0.1 117000. 118000. 1000.
0.1 0.1 0.1 29250. 59000. 1000.
29250. 0.1 0.1 58500. 59000. 1000.
58500. 0.1 0.1 87750. 59000. 1000.
87750. 0.1 0.1 117000. 59000. 1000.
* number of icon records
5
1 f18
2 mig1
3 missile
4 tank
5 truck
* END OF OBJECT
```

E.4.2 Map File.

```
*****************************************************************************
* FILE: battlesim13.map
* AUTHOR: Capt Seth Guanu
* DATE: 17 Nov 93
* DESCRIPTION:  This file contains the battlesim sector-to-LP description
* for benchmark scenario 13. The BATTLESIM application must
* be executed with 8 LPs to use this particular map file.
*               Each line describes a single mapping from a single sector ID
* to a single LP id, in that order.  There should be exactly
* as many lines as there are sectors from the scenario file(s)
* being used for a given simulation run.
*
*****************************************************************************
* Mapping sector 1 to LP 0
1 0
* Mapping sector 2 to LP 1
2 1
* Mapping sector 3 to LP 2
3 2
* Mapping sector 4 to LP 3
4 3
* Mapping sector 5 to LP 4
5 4
* Mapping sector 6 to LP 5
6 5
* Mapping sector 7 to LP 6
7 6
* Mapping sector 8 to LP 7
8 7
```

E.4.3 Scenario Diagram. This diagram depicts the movement of all eight players in benchmark 13 throughout the course of the scenario.

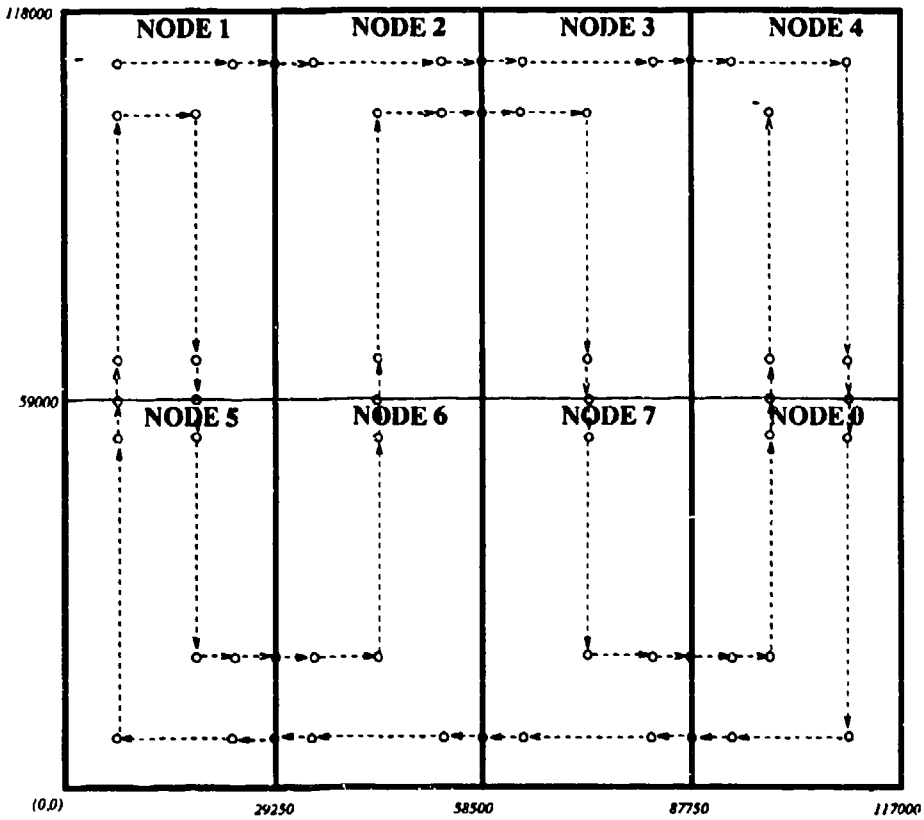

Figure 16. Benchmark Scenario 13

Appendix F. BATTLESIM Configuration Guide

F.1 Software Files.

The software files supporting the current version of BATTLESIM, listed in alphabetical order for quick reference, include the following:

- app_stru.h
- application.h
- battle.c and battle.h
- bs_player.c and bs_player.h
- cube2.c and cube2.h
- dll.c
- ex_event.c and ex_event.h
- filters.c
- globals.h
- icon.c
- init_maps.c
- interfaceB.c and interfaceB.h
- ll.c and ll.h
- lp_man.c
- map.c and map.h
- methods.c
- myfilters.c
- object_mgr.c and object_mgr.h
- player.c and player.h
- playerset.c and playerset.h
- predict.c and predict.h
- proc1.arcs
- protocol.c
- route.c and route.h
- rt_pt.c and rt_pt.h
- sector.c and sector.h
- sim_func.c and sim_func.h
- sim_read.c and sim_read.h
- sim_stru.h
- tchmap.c

- terrain.c
- use_visit.c
- scenario files
- schedule.c and schedule.h
- map files
- Makefile

The functional description of each of these software files is provided in the next section.

F.2 Functional Description.

application.h - This file contains conditional compilation 'defines' which pass on information to BATTLESIM. Fields of interest include:

- NUM_PROCS - the number of LPs, not nodes, which a run will use. This value *must* match the number of LPs specified to the 'host' program when executing a scenario, or else the run will abnormally terminate.
- INPUT_ARCS - the file specifying the SPECTRUM communication arcs between BATTLESIM LPs. A file must be specified, even though it is not actively used, because SPECTRUM requires it.
- MAXTIME - specifies a run's maximum allowable execution time in seconds.

app_stru.h - Contains structure definitions common to the use of the Battlesim subclass.

battle.c - `Battle.c`, formerly known as `rizsim.c`. It contains the functions necessary for the simulation driver to interface with the application. The following function are performed in this file:

- simulation initialization
- scheduling initial player events
- starting and stopping the user's screen output

battle.h - Header file for battle.c.

bs_player.c and bs_player.h - The files containing the object-based definition of a BATTLESIM player subclass. All the methods necessary to retrieve, modify, and delete all the fields in a Battlesim player subclass without knowledge of the player's underlying structure are kept here.

cube2.c - This file contains all the Intel Hypercube specific functions necessary to support BATTLESIM. It is the lowest level of communications between LPs in the 'layered approach' implemented by BATTLESIM. Soderhoim's message-passing protocol — comprised of the functions node_send_one_message and node_get_one_message — is kept in here. His approach bypassed SPECTRUM to make direct calls to the Hypercube.

cube2.h - This file defines Intel message types, and contains a table definition which maps the LP identifiers to the Hypercube node and process numbers. This tells each node process how to send a message to any other LP.

dll.c - This package is very similar to Capt Rizza's linked list package, except this package was designed by Capt Soderholm to support doubly linked lists, i.e. linked lists with both head and tail pointers so it can be traversed in either direction.

ex_event.c - Contains the *execute_event* method for each of the simulation events and the *execute_event* function that determines which of them are used base on the type of the event passed in..

ex_event.h - Contains the numeric identifiers for Event Object types.

filters.c - another SPECTRUM file supposedly describing the various time synchronization protocols used by BATTLESIM. There currently are none. The only SPECTRUM file that appears to reference this file is the Makefile itself, but it is kept because the 'user interface' utility builds this file for any provided filter set (12). The actual filter file for BATTLESIM is `myfilters.c`.

globals.h - SPECTRUM supports only one type of message for transferring information from one LP to another. This event-type structure definition is stored within this file, and contains at least the following fields:

1. a message time-stamp
2. the event type
3. line number over which it is sent
4. identifiers of the source and destination LPs (12)

While additional state information fields specific to a given application can be added as long as no existing fields are changed or deleted, BATTLESIM currently adds no fields of its own. However, since this definition is used throughout SPECTRUM, all SPECTRUM software files should be recompiled if this file is modified in any manner.

icon.c - An object-based icon management package developed by Mr. Rick Norris. It utilizes a linked list to hold all the icons used to represent players in Battlesim, tracking both their number and name.

init_maps.c - Contains the method *init_Maps* that initializes the maps in the Object Relationship Map object in *map.c*.

interfaceB.c - The BATTLESIM version of the SPECTRUM file `interface1.c`. This file acts as the "link" between the BATTLESIM application and the lower-level SPECTRUM and Hypercube-specific functions, by containing all calls to the encapsulated lower-level structures (12:12).

interfaceB.h - Contains the function protocols for methods that are available in *interfaceB.c*.

ll.c and ll.h - A linked list package designed to support LIFO, FIFO, and priority queues containing any kind of data structure desired. This package is used extensively to build and maintain several BATTLESIM structures, including the six linked lists contained within the 'player'

definition and the buckets in the playerset's hash table. The file `ll.h` contains the declarations for the functions in `ll.c`.

lp_man.c - Contains the code implementing the SPECTRUM logical process manager, which is both application and machine independent. The LP manager maintains the input queue of messages from other LPs in simulation time-stamp order. BATTLESIM only uses the initialization functions.

map.c and map.h - Contain the object-based definition of the Relationship Map object.

methods.c - Contains the polymorphic methods used to access player subclasses. It provides control over which exact method is called based on the type of the Player Object.

myfilters.c - the SPECTRUM file which actually holds any 'filters' used. There are currently no filters used by BATTLESIM.

object_mgr.c and object_mgr.h - Contain the methods required to manage the updating of player object copies on remote LPs.

player.c and player.h - The files containing the object-based definition of a Player Object. All the methods necessary to retrieve, modify, and delete all the fields in a Player Object without knowledge of the player's underlying structure are kept here.

predict.c and predict.h - Contain the object-based definition of the Event Predictor Object.

proc1.arcs - LPs in SPECTRUM communicate via unidirectional lines known as *arcs*. This file tells SPECTRUM which LPs communicate with each other, i.e. it describes BATTLESIM's communications 'network'. Even though this file contains no entries which are actively used, SPECTRUM still requires it to exist.

playerset.c and playerset.h - The files which contain the object-based definition of a BATTLESIM playerset, presently implemented as an open hash table with buckets composed of linked lists. All the methods necessary to retrieve, modify, and delete the playerset without knowledge

of the playerset's underlying structure are stored here. The user should refer to Appendix B for a complete listing and description of these methods.

protocol.c - This file implements the conservative time synchronization algorithm used by BATTLESIM. The components previously used to support the optimistic time synchronization protocol have been removed.

route.c and route.h - Contain the object-based definition of a route.

rt_pt.c and rt_pt.h - Contain the object-based definition of a route point.

schedule.c and schedule.h - Contain the object-based definition of the Event Scheduler.

sector.c and sector.h - The two files which hold the object-based definition of a BATTLESIM sector. The sector "container" object, a 64-entry entry, is also here. All the methods used to retrieve, modify, and delete the fields in the sector via the sector array without knowledge of the underlying structures are kept here. The user can see a complete listing of all sector methods, along with their associated descriptions, in Appendix B.

sim_func.c and sim_func.h - One of two main BATTLESIM application files. It contains application-specific functions which are independent of the Hypercube and SPECTRUM.

sim_read.c and sim_read.h - These files contain the functions which read the data from an LPs scenario file, and store it in the appropriate location. For route points, it reverses those read so they are stored in reverse order as required.

sim_stru.h - This file contains the structural definition of a BATTLESIM player.

tchmap.c - A TCHSIM file which contains an object-based implementation of an object-to-LP map. Each map consists of a set of object instances to logical processes. This is used by BATTLESIM to track sector-to-LP assignments.

terrain.c - An object-based implementation of a terrain file, used to let BATTLESIM know certain required battlefield characteristics like minimum and maximum battlefield coordinates.

use_visit.c - This file acts as an interface to VISIT, the visual graphics driver program designed by Capt DeRouchey to display graphics output files created by BATTLESIM. Specifically, it contains functions which generate records in the grap cs file to start VISIT, stop VISIT, and change the visual status of players.

scenario files - These files, whose names end with a .in5 extension, are used to convey battlefield and player state information associated with a given battlefield scenario. Each LP must read a scenario file during initialization. That scenario file may be designed for use by only one LP, or may in fact be shared by multiple LPs.

map files - Each BATTLESIM scenario requires that a map file, whose name begins with the .map extension, be provided describing the sector-to-LP assignments. Since this assignment is static for the duration of the scenario, so any player entering a sector owned by a given LP is controlled by that LP while in the sector.

Makefile - This file provides an automated means of compiling and linking all the software files necessary to execute BATTLESIM. The current Makefile is contained in Appendix E.

F.3 Makefile

The following Makefile shows the compilation order and dependencies of the Battlesim code.

```
################## Makefile for BATTLESIM ######################
# Make battlesim by Hartrum
# 08/12/93
#
# 01/31/92 - Now using standard /usr/simulate/spectrum/afit/cube2.h
# 04/28/92 - Added sim_read.o
# 05/04/92 - Replaced neq1.o/neq1.c with neq_sod.o/neq_sod.c
# 05/04/92 - Removed event.c - it was unused.
# 06/24/92 - Added interfacex.c and tchsim/clock.o
# 07/10/92 - Added BATTLEPATH and BATTLEOBJS, changed to new baseline..
# 07/13/92 - Added icon.c & terrain.c to battlesim; changed executable
# name from "rizsim" to "battlesim"
# 07/14/92 - Changed cube2.h path to spectrum
# 07/16/92 - Added soderstuff.c
# 07/17/92 - Modified path to use baseline host2.c
# Removed rollback.* and soderstuff.*
# Changed rizsim.* to battle.*
# Changed interfacex.* to interfaceS.*
# 07/25/92 - Removed events.c
# Added playerset.c
```

```
# 7/30/92  - Bergman replaced all references of interfaceS to interfaceB
#            (interfaceB = interface1 + spectrum calls)
# 7/30/92  - Bergman removed all references to -DSODER_PROTOCOL
#            (to bypass round-robin aircraft-LP allocation and msg passing)
# 7/30/92  - Bergman removed protocol.c/protocol.o entirely, and removed
#            existing references to protocol.h (none)
# 7/30/92  - NOTE: This architecture should be run with MULTIPLE input files
#            to ensure each node loads only what it is supposed to have!
# 8/04/92  - Bergman added player.c to file
# 1/11/93  - Bergman/VanHorn changed myfilters.c/o to chanclocks.c/o,
#            and removed filters.c (this file now incorporated into each
#            individual filter file)
# 7/10/93  - Hartrum - Changed chanclocks.c to tchsimclocks.c
# 7/22/93  - Hartrum - Changed to link tchmap.o from TCHPATH instead of
# compiling it.
# 8/12/93  - Hartrum - Changed include path for use_visit to VISITINCL.
# 10/01/93 - Trachsel - added map.c, ex_event.c, init_maps.c, predict.c,
# schedule.c, removed sensor.c, modified battle,event,sector,sim_func,
# sim_read

SpecOBJS = cube2.o lp_man.o tchsimclocks.o
TCHOBJS = tchmap.o neqA.o event.o sim_cntrl.o
RizOBJS  = dll.o ll.o
BATTLEOBJS  = sim_func.o sim_read.o battle.o terrain.o icon.o use_visit.o
player.o playerset.o sector.o map.o ex_event.o init_maps.o predict.o
schedule.o route.o rt_pt.o methods.o bs_player.o object_mgr.o
SODEROBJS  = interfaceB.o
RizLIB = -lm

AFITPATH = /usr/simulate/spectrum/afit
AFITINCL = /usr/simulate/spectrum/afit/include
BATTLEPATH = /usr/simulate/battlesim/source
BATTLEINCL = /usr/simulate/battlesim/source/include
BERGPATH = /usr2/eng/kbergman/batlsim
FILTERPATH = /usr/simulate/spectrum/filters
NEWPATH = /usr/simulate/battlesim/new
RIZPATH = /usr/simulate/rizsim
SODPATH = /usr/simulate/rizsim/soderholm
TCHPATH = /usr/simulate/tchsim
TCHINCL = /usr/simulate/tchsim/include
UVAPATH = /usr/simulate/spectrum/uva
UVAOLDPATH = /usr/simulate/spectrum/uva/old
VISITPATH = /usr/simulate/cubevisit/source
VISITINCL = /usr/simulate/cubevisit/source/include
WGTPATH = /usr2/eng/wtrachse/thesis/src
NEWDUG = /usr/simulate/battlesim/new/looney
WORKDUG = /usr2/eng/dlooney/bin
BATOBJS = battoge.o # simutils.o
DEFCOMM = -DDEBUG_ON -DMAIN_CODE
BINPATH = /usr2/eng/dlooney/bin
DUGPATH = /usr2/eng/dlooney/cube/source
DCLPATH = /usr2/eng/dlooney/tchsim
VISITINCL = /usr2/eng/dlooney/hostdir/include
VISITPATH = /usr2/eng/dlooney/hostdir
IPSC2LIBS =  -lsocket /usr/local/lib/AFIT_comc.a
IPSC2FLAGS = -DIPSC2 -I/usr/include -I$(VISITINCL)

all: visithost battlesim

visithost: $(BATOBJS) ${VISITPATH}/host3.c
cc -o visithost -I${WGTPATH} -I$(VISITINCL) -I$(AFITINCL) $(BATOBJS)
${VISITPATH}/host3.c $(IPSC2LIBS) -host

battlesim: $(RizOBJS) ${TCHPATH}/simdrive.o tchmap.o ${TCHPATH}/clock.o
${WORKDUG}/neqA.o $(SpecOBJS) $(BATTLEOBJS) ${SODEROBJS} event.o sim_cntrl.o
```

```
cc -o battlesim $(RixOBJS) ${TCHPATH}/simdrive.o tchmap.o
${TCHPATH}/clock.o $(WORKDUG)/neqA.o $(SpecOBJS) sim_cntrl.o
$(RixLIB) $(BATTLEOBJS) ${SODEROBJS} event.o -node

sim_func.o: ${WGTPATH}/sim_stru.h ${BATTLEINCL}/ll.h ${WGTPATH}/sim_func.c
${WGTPATH}/battle.h ${WGTPATH}/application.h ${WGTPATH}/map.h
${WGTPATH}/globals.h
cc -c -I${WGTPATH} -I${BATTLEINCL} ${WGTPATH}/sim_func.c
sim_read.o: ${WGTPATH}/sim_stru.h ${WGTPATH}/player.h ${BATTLEINCL}/route_pt.h
${BATTLEINCL}/sim_read.h ${WGTPATH}/sim_read.c ${WGTPATH}/application.h
${WGTPATH}/globals.h
cc -c -I${WGTPATH} -I${BATTLEINCL} -I${SODPATH} ${WGTPATH}/sim_read.c
battle.o: ${WGTPATH}/sim_stru.h ${WGTPATH}/battle.h ${WGTPATH}/interfaceB.h
${WGTPATH}/ex_event.h ${WGTPATH}/application.h ${WGTPATH}/AFITcom.h
${WGTPATH}/battle.c
cc -c -I${WGTPATH} -I${BATTLEINCL} ${WGTPATH}/battle.c
terrain.o: ${BATTLEPATH}/terrain.c
cc -c ${BATTLEPATH}/terrain.c

icon.o: ${BATTLEPATH}/icon.c
cc -c ${BATTLEPATH}/icon.c

sector.o: ${WGTPATH}/{map.h} ${WGTPATH}/sim_stru.h ${BATTLEINCL}/route_pt.h
${WGTPATH}/sector.h ${WGTPATH}/sector.c
cc -c -I${WGTPATH} -I${BATTLEINCL} ${WGTPATH}/sector.c

player.o: ${WGTPATH}/sim_stru.h ${WGTPATH}/methods.h ${WGTPATH}/player.h
${WGTPATH}/player.c
cc -c -I${WGTPATH} -I${BATTLEINCL} -I${SODPATH} ${WGTPATH}/player.c

bs_player.o: ${WGTPATH}/sim_stru.h ${WGTPATH}/app_stru.h ${WGTPATH}/bs_player.h
${WGTPATH}/route.h ${WGTPATH}/ex_event.h ${WGTPATH}/player.h
${WGTPATH}/bs_player.c
cc -c -I${WGTPATE} -I${BATTLEINCL} -I${SODPATH} ${WGTPATH}/bs_player.c

methods.o: ${WGTPATH}/player.h ${WGTPATH}/bs_player.h ${WGTPATH}/methods.c
cc -c -I${WGTPATH} ${WGTPATH}/methods.c

object_mgr.o: ${WGTPATH}/player.h ${WGTPATH}/interfaceB.h ${WGTPATH}/map.h
${WGTPATH}/ex_event.h ${WGTPATH}/object_mgr.c
cc -c -I${WGTPATH} -I${BATTLEINCL} -I${SODPATH} ${WGTPATH}/object_mgr.c

route.o: ${WGTPATH}/route.h ${WGTPATH}/player.h ${BATTLEINCL}/sim_read.h
${WGTPATH}/route.c ${WGTPATH}/rt_pt.h
cc -c -I${WGTPATH} -I${BATTLEINCL} ${WGTPATH}/route.c

rt_pt.o: ${WGTPATH}/rt_pt.h ${WGTPATH}/rt_pt.c
cc -c -I${WGTPATH} ${WGTPATH}/rt_pt.c

ll.o: ${WGTPATH}/ll.h ${BATTLEINCL}/route_pt.h ${WGTPATH}/ll.c
cc -c -I${WGTPATH} -I${BATTLEINCL} -I${SODPATH} ${WGTPATH}/ll.c

map.o: ${WGTPATH}/ll.h ${WGTPATH}/map.h ${WGTPATH}/map.c
cc -c -I${WGTPATH} ${W3TPATH}/map.c

init_maps.o: ${WGTPATH}/ex_event.h ${WGTPATH}/predict.h ${WGTPATH}/player.h
${WGTPATH}/map.h ${WGTPATH}/init_maps.c
cc -c -I${WGTPATH} -I${BATTLEINCL} ${WGTPATH}/init_maps.c

ex_event.o: ${WGTPATH}/sim_stru.h ${WGTPATH}/player.h ${WGTPATH}/playerset.h
${BATTLEINCL}/sector.h ${WGTPATH}/sim_func.h ${BATTLEINCL}/message.h
${WGTPATH}/ex_event.h ${WGTPATH}/AFITcom.h ${WGTPATH}/map.h
${WGTPATH}/ex_event.c
cc -c -I${WGTPATH} -I${BATTLEINCL} ${WGTPATH}/ex_event.c
```

```
predict.o: ${WGTPATH}/player.h ${WGTPATF}/playerset.h ${BATTLEINCL}/sector.h
${WGTPATH}/sim_func.h ${WGTPATH}/interfaceB.h ${BATTLEINCL}/route_pt.h
${WGTPATH}/ex_event.h ${WGTPATH}/predict.h ${WGTPATH}/map.h
${WGTPATH}/predict.c
cc -c -I${WGTPATH} -I${BATTLEINCL} ${WGTPATH}/predict.c

schedule.o: ${WGTPATH}/player.h ${WGTPATH}/battle.h ${WGTPATH}/map.h
${WGTPATH}/interfaceB.h ${WGTPATH}/schedule.c
cc -c -I${WGTPATH} -I${BATTLEINCL} ${WGTPATH}/schedule.c

cube2.o: ${AFITPATH}/cube2.c ${WGTPATH}/application.h ${WGTPATH}/globals.h
${AFITINCL}/cube2.h
cc -c -I${WGTPATH} -I${BATTLEINCL} -I${AFITINCL} ${AFITPATH}/cube2.c

event.o: ${WGTPATH}/event.h ${WGTPATH}/event.c
cc -c -I${WGTPATH} ${WGTPATH}/event.c

battoge.o:
cc $(IPSC2FLAGS) -c -I${WGTPATH} $(VISITPATH)/battoge.c $(IPSC2LIBS)


use_visit.o: ${BATTLEINCL}/sim_stru.h ${BATTLEINCL}/battle.h
${DUGPATH}/use_visit.c
cc $(IPSC2FLAGS) -c -I${WGTPATH} -I${BATTLEINCL} -I${VISITINCL}
${DUGPATH}/use_visit.c


lp_man.o:${DCLPATH}/lp_man.c ${WGTPATH}/application.h ${WGTPATH}/globals.h
cc -c -I${WGTPATH} -I${BATTLEINCL} ${DCLPATH}/lp_man.c

tchsimclocks.o: ${DUGPATH}/tchsimclocks.c ${BATTLEINCL}/application.h
${BATTLEINCL}/globals.h
cc -c -I${WGTPATH} -I${BATTLEINCL} ${DUGPATH}/tchsimclocks.c

playerset.o: ${BATTLEINCL}/route_pt.h ${BATTLEINCL}/sim_stru.h
${WGTPATH}/playerset.h ${WGTPATH}/playerset.c
cc -c -I${WGTPATH} -I${BATTLEINCL} -I${SODPATH} ${WGTPATH}/playerset.c


# SODEROBJS -
# I modified these TCHSIM OBJECTS -

neqA.o: ${DCLPATH}/neqA.c
cc -c ${DCLPATH}/neqA.c

tchmap.o: ${DCLPATH}/tchmap.c
cc -c ${DCLPATH}/tchmap.c

sim_cntrl.o: ${BATTLEINCL}/ll.h  ${BINPATH}/ll.o  ${SODEROBJS}
${DUGPATH}/sim_cntrl.c
cc -c -I${WGTPATH} -I${BATTLEINCL} -I${VISITINCL} ${DUGPATH}/sim_cntrl.c


# SODEROBJS -

interfaceB.o: ${TCHINCL}/tchsim.h ${WGTPATH}/application.h ${WGTPATH}/globals.h
${WGTPATH}/event.h ${BATTLEINCL}/message.h ${WGTPATH}/player.h
${WGTPATH}/ex_event.h ${WGTPATH}/interfaceB.c
cc -c -I${WGTPATH} -I${TCHINCL} -I${BATTLEINCL}
-I/usr/fac/hartrum/tchsim/ver2 ${WGTPATH}/interfaceB.c

dll.o: ${SODPATH}/dll.c
cc -c -I${SODPATH} ${SODPATH}/dll.c
```

Bibliography

1. (ATWG), Architectural Technical Working Group. *Software Development Plan (SDP) Volume II, Software Structural Model (SSM) Design Methodology for the Modeling Library Components for the Joint Modeling & Simulation System (J-MASS) Program Version 2.0*. Software Development Plan J-MASS-SSM-2.0, The Joint Modeling & Simulaiton System (J-MASS) Program, Feb 1993.

2. Bergman, Kenneth C. *Spatial Partitioning of a Battlefield Parallel Discrete Event Simulation*. MS thesis AFIT/GCS/ENG/92D-03, Air Force Institute of Technology, 1992.

3. Bischak, Diane P. and Stephen B. Roberts. "Object-Oriented Simulation." *Proceedings of the 1991 Winter Simulation Conference*. 194–203. New York: IEEE Press, 1991.

4. Booch, Grady. *Object Oriented Design With Applications*. The Benjamin/Cummings Publishing Company, Inc., 1991.

5. Chandy, K. M. and Jayadev Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, *SE-5*(5):440–452 (Sep 1979).

6. Corey. Philip D. and John R. Clymer. "Discrete event simulation of object mvoement and interactions," *Simulation*, *56*(3):167–174 (March 1991).

7. DeRouchey. Capt William J. *A Remote Visual Interface Tool for Simulation control and Display*. MS thesis AFIT/GCS/ENG/90D-03, Air Force Institute of Technology, 1990.

8. Donald P. Duckett, Jr. *The Application of Statistical Estimation Techniques Terrain Modeling*. MS thesis AFIT/GCE/ENG/91D-02, Air Force Institute of Technology, 1991.

9. Drolet, Jocelyn R. and Colin L. Moodie. "Object-Oriented Simulation with Smalltalk-80: A Case Study." *Proceedings of the 1991 Winter Simulation Conference*. 312–322. New York: IEEE Press, 1991.

10. Evans, John B. *Structures of Discrete Event Simulation*. Ellis Horwood Limited, 1988.

11. Guttman, Michael, et al. "A Methodology for Developing Distributed Applicaitons," *Object Magazine*, 55–59 (January-February 1993).

12. Hartrum, Thomas C. *AFIT Guide to SPECTRUM*. Unpublished Report, Air Force Institute of Technology, Feb 1992.

13. Hartrum, Thomas C. *TCHSIM: A Simulation Environment for Parallel Discrete Event Simulation*. Unpublished Report, Air Force Institute of Technology, Jan 1992.

14. Jefferson, David R. "Virtual Time," *ACM Transactions on Programming Languages and Systems*, *3*(7):404–425 (Jul 1985).

15. Lee, Kenneth J., et al. *Model-Based Software Development*. Software Architecture Engineering Project CMU/SEI-92-SR-00, ESD-92-SR-00, Software Engineering Institute, Carnegie Mellon University, Dec 1991.

16. Matsche, John J. "Object-Oriented Programming in Standard C," *Object Magazine*, 71–73 (January-February 1993).

17. Mattos, Nelson M., et al. "Grand Tour of Concepts for Object-Orientation from a Database Point of View," *Data & Knowledge Engineering*, *9*:321–352 (1992/93).

18. Moser, Robert S. *A Spatially Partitioned Parallel Simulation of Colliding Objects*. MS thesis AFIT/GCS/ENG/91D-15, Air Force Institute of Technology, 1991.

19. Mullin, Mark. *Object Oriented Program Design*. Addison-Wesley Publishing Company, Inc., 1989.

20. Palmer, John. "Mysterious and Miraculous Message Routing," *Object Magazine*, 34,72 (May-June 1993).

21. Reynolds, Paul F. "A Spectrum of Options for Parallel Simulation." *Proceedings of the 1988 Winter Simulation Conference*. 325-332. Dec 1988.

22. Reynolds, Paul F. "SRADS with Local Rollback." *Proceedings of the SCS Multi-conference on Distributed Simulation 22*. 161-164. Jan 1990.

23. Rizza, Robert J. *An Object-Oriented Military Simulation Baseline for Parallel Simulation Research*. MS thesis AFIT/GCS/ENG/90D-12, Air Force Institute of Technology, 1990.

24. Roberts, LeeAnne. *A Software System to Create a Hierarchical, Multiple Level of Detail Terrain Model*. MS thesis AFIT/GCS/ENG/88D-13, Air Force Institute of Technology, 1988.

25. Rumbaugh, James, et al. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

26. Sherry, Christine M. *Object-Oriented Analysis and Design of the Saber Wargame*. MS thesis AFIT/GCS/ENG/91D-21, Air Force Institute of Technology, 1991.

27. Shewchuk, John P. and Tien-Chien Chang. "An Approach to Object-Oriented Discrete-Event Simulation of Manufacturing." *Proceedings of the 1991 Winter Simulation Conference*. 302-311. New York: IEEE Press, 1991.

28. Simpson, Dennis Joseph. *An Application of the Object-Oriented Paradigm to a Flight Simulator*. MS thesis AFIT/GCS/ENG/91D-22, Air Force Institute of Technology, 1991.

29. Smith, David A. "Virtual Reality: Redefining the meaning of human-computer interaction," *Object Magazine*, 65-68 (September-October 1992).

30. Soderholm, Capt S. R. *A Hybrid Approach to Battlefield Parallel Discrete Event Simulation*. MS thesis AFIT/GCS/ENG/91D-23, Air Force Institute of Technology, 1991.

31. Sweeney, Paula, et al. "Modelling Physical Objects for Simulation." *Proceedings of the 1991 Winter Simulation Conference*. 1187-1193. New York: IEEE Press, 1991.

32. Zeigler, Bernard P. *Object-Oriented Simulation with Hierarchical, Modular Models*. Academic Press, Inc., Harcourt Brace Jovanovich, Publishers, 1990.

33. Zyda, Michael J., et al. "NPSNET: Hierarchical Data Structures For Real-Time Three-Dimensional Visual Simulation," *Computers and Graphics*, *17*(1):66-69 (1993).

34. Zyda, Michael J., et al. *NPSNET: Constructing A 3D Virtual World*. Technical Report, Naval Post Graduate School, Department of Computer Science, 1992.

Vita

Captain Walter G. Trachsel was born February 22, 1964, in Elgin, Illinois. After graduating from Prospect High School in 1982, he enrolled in the University of Illinois and graduated with a Bachelor of Science in Computer Science. After graduation, he enrolled in the Air Force's Officer Training School and was commissioned as a second lieutenant in July of 1987.

Captain Trachsel's first assignment was with the 1912th Computer Systems Group at Langley Air Force Base as a software test engineer for the Tactical Air Control Systems' (TACS) Message Processing Center and Control and Reporting Center. He was responsible for testing joint interoperability of the TACS systems with similar systems of the Army, Navy, and Marines. Captain Trachsel's subsequent position with the 1912th was as Chief of the Graphics Applications section for the Contingency TACS Automated Planning System (CTAPS).

While stationed at Langley AFB, Captain Trachsel earned a Masters of Business Administration in Management Information Systems from Golden Gate University. Following this, he entered AFIT in May of 1992.

Permanent address: 300 S. Louis Street
Mt Prospect, IL 60056

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE
Object Interaction in a Parallel
Object-Oriented Discrete-Event Simulation

5. FUNDING NUMBERS

6. AUTHOR(S)
Walter G. Trachsel, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/93D-22

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Capt Rick Painter
2241 Avionics Circle, Suite 16
WL/AAWA-1 BLD 620
Wright-Patterson AFB, OH 45433-7765
(513)-255-4429

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis investigates object interaction issues involved in developing an object-oriented parallel discrete-event simulation and develops a simulation model that provides object interaction capabilities. The research covers issues in object representation, object interaction, object management, discrete-event simulation, and parallel simulation. There are three primary types of objects that the research discusses. The first type is a basic simulation object, whose size and behavior is insignificant compared to the size of the simulation as a whole. The second type is an aggregate object which consists of smaller component objects that interact and affect the performance of the larger object as a whole. Finally, there are environmental objects, such as terrain and weather, whose size and impact are significant to the entire simulation environment. This research addresses issues about whether these objects can be represented the same in the simulation or whether a special case must exist for each type of object.

As a result of the research, a simulation model is designed that allows interaction between simulation objects. The design goals used to develop the model were strict use of object-oriented practices, ease of simulation modification, and reuse across other applications in the same class of simulations. In this case, the class of simulations is interacting objects in a spatially partitioned discrete-event simulation.

The model is then implemented using AFIT's Battlesim program, which is a parallel discrete-event simulation that has a battlefield divided into partitions and allows objects to move and interact.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Object-Oriented, Discrete-Event, Parallel, Simulation	131
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL